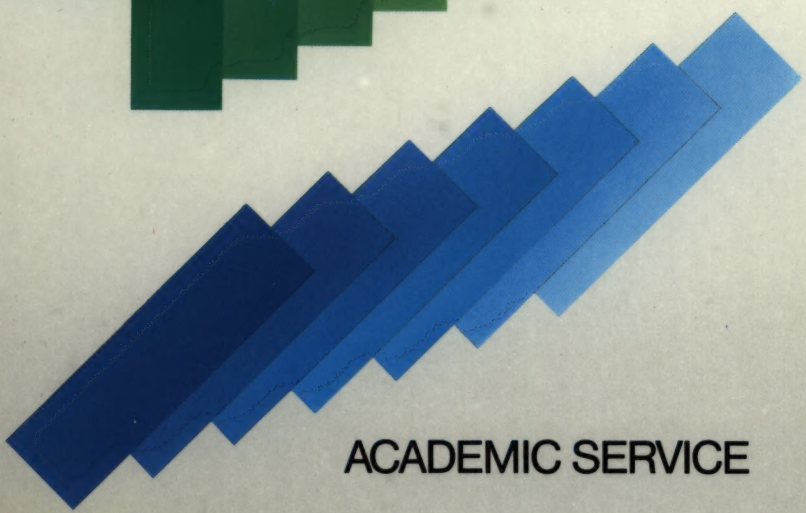
HENK VAN DER VORST

PARALLEL REKENEN EN SUPER COMPUTERS



ACADEMIC SERVICE

Parallel rekenen en supercomputers

HENK VAN DER VORST

PARALLEL REKENEN EN SUPER COMPUTERS

ACADEMIC SERVICE

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Vorst, Henk A. v.d.

Parallel rekenen en supercomputers / Henk A. van der Vorst. -
Schoonhoven : Academic Service. - Ill.
Met lit. opg.
ISBN 90-6233-300-1
SISO 664 SVS 8.12.3 UDC 681.31 NUGI 853
Trefw.: computers.

10 9 8 7 6 5 4 3 2 1

Uitgegeven door: Academic Service
Postbus 81
2870 AB Schoonhoven
Zetwerk: klaas waagmeester, text & design, Hantum (op Macintosh II met PageMaker 3.0)
Omslag: Olga van der Vorst / FennArt, Steenwijk
Druk: Krips Repro Meppel
Bindwerk: Meeuwis, Amsterdam

ISBN 90 6233 300 1
NUGI 853

INHOUD

INLEIDING

Het is al geruime tijd duidelijk dat het klassieke 'Von Neumann' concept niet meer toereikend is om de voor wetenschappelijke toepassingen gewenste snelheid van computers te bereiken. Dit heeft krachtige impulsen gegeven aan het onderzoek naar vormen van parallelle verwerking.

Bij het parallel verwerken van rekenprogramma's kan men denken aan het simultaan uitvoeren van grote onafhankelijke gedeelten van een berekening (grofschalig parallellisme), echter men kan ook denken aan het gelijktijdig uitvoeren van zeer kleine operaties zoals de elementaire operaties waaruit de optelling van twee reële getallen is samengesteld (zeer fijnschalig parallellisme). Deze laatste vorm van parallellisme is sedert de introductie van de vectorsupercomputer CRAY-1 in 1976 tot grote bloei gekomen en alle huidige supercomputers maken daarvan gebruik. Het in deeloperaties opsplitsen van wiskundige operaties als de optelling en de vermenigvuldiging duidt men wel aan als segmentatie. De functional units die zo'n gesegmenteerde optelling of vermenigvuldiging kunnen uitvoeren heten dan gesegmenteerde functional units. De computers die op basis hiervan de rekensnelheid kunnen opvoeren heten, om redenen die we later duidelijk zullen maken, vector- of pipeline processoren (of kortweg vectorcomputers). We zullen in dit boek het principe van het op dit soort computers toegesneden vector-rekenen in detail toelichten.

Door het toepassen van segmentatie kon een grote sprong voorwaarts gemaakt worden. De klassieke mainframes in de jaren '70 haalden op hun best rekensnelheden in de orde van 1 miljoen floating point operaties (lees hiervoor maar een optelling, aftrekking of vermenigvuldiging (maar geen deling) van twee reële getallen) per seconde. Door het simultaan uitvoeren van de deelbewerkingen van deze operaties kunnen in sommige ideale omstandigheden rekensnelheden van circa 1000 miljoen floating point operaties worden gehaald (bijvoorbeeld door de NEC SX-2). Enerzijds zijn tegenwoordig ook deze rekensnelheden nog onvoldoende hoog en anderzijds kan bij veel belang-

rijke toepassingen het principe van vector-rekenen niet of onvoldoende benut worden bij gebrek aan voldoende fijnschalig parallellisme. Daarom is het zoeken naar praktisch realiseerbare vormen van grofschalige parallelle verwerking met grote energie voortgezet. De eerste grote supercomputer die hiervan een bescheiden gebruik maakte was de CRAY X-MP. Op deze computer, die in feite bestaat uit vier nauw met elkaar verbonden (verbeterde) CRAY-1's, kunnen grote onderdelen van een rekenproces simultaan worden uitgevoerd, waardoor soms een reductie in de verwerkingstijd met een factor 4 gerealiseerd kan worden. Elk van deze onderdelen kan dan weer de vectormogelijkheden van de aparte processoren benutten.

Vier aan elkaar gekoppelde processoren staan nog maar een bescheiden vorm van parallelle verwerking toe. De problemen die men moet overwinnen om grotere aantallen efficiënt aan elkaar te koppelen zijn echter zeer groot. Ook is het nog niet erg duidelijk onder welke omstandigheden grote rekenprocessen hiervan kunnen profiteren. Er zijn inmiddels talrijke ontwerpen gemaakt en gerealiseerd van computersystemen met een groot aantal processoren en er tekent zich een zekere voorkeur voor bepaalde architecturen af. Daardoor kon met enig succes het parallel rekenen commercieel exploiteerbaar worden en de eerste ervaringen duiden erop dat grofschalig parallellisme inderdaad een grote bijdrage kan leveren bij het realiseren van zeer korte verwerkingstijden.

Het ziet er naar uit dat de krachtigste supercomputers in de voorzienbare toekomst allemaal gebaseerd zijn op een combinatie van grofschalig parallellisme (groot aantal parallelle processoren) en fijnschalig parallellisme (segmentatie van de functional units in elke processor afzonderlijk). Bovendien leert de ervaring dat processen die zich goed lenen voor efficiënte vector-verwerking (fijnschalig parallellisme) zich doorgaans ook goed lenen voor grofschalige parallelle verwerking. Om deze redenen is bij de opzet van dit boek gekozen om uit te gaan van een vrij uitvoerige behandeling van het vector-rekenen. Hierbij kon gebruik gemaakt worden van een inmiddels vrij ruime ervaring op dit gebied.

Nadat men in het eerste deel van het boek vertrouwd is geraakt met de principes van segmentatie, de mogelijkheden van vectorverwerking en de moeilijkheden daarbij, komen we op vrij natuurlijke wijze terecht bij het grofschalige parallellisme. Dit grofschaliger parallellisme, het echte parallel processing of parallel rekenen, wordt ietwat globaler behandeld. Voor deze globalere behandeling is gekozen omdat het ernaar uitziet dat er op dit gebied zowel qua architectuur als wat betreft programmeertalen en compilers nog wel het een en ander zal veranderen. We hebben daarom getracht zoveel mogelijk de algemene lijnen in het oog te houden en ons zo min mogelijk af te laten leiden door technische details. Overigens zijn de beschouwingen in dit deel ook weer geïllustreerd met ervaringen opgedaan op echte parallelle computers.

In het hele boek staan de toepassingen bij het technisch wetenschappelijk rekenwerk centraal. Dat houdt in dat architecturen die hiervoor nog niet zo geschikt zijn gebleken best een heel grote waarde kunnen hebben voor andere probleemklassen, zoals bijvoorbeeld database management of artificial intelligence problemen. Ook zouden veranderingen in de wiskundige modellering van fysische verschijnselen kunnen leiden tot geheel andere algoritmen waarvoor andere dan de in dit boek besproken computers zinvol zouden kunnen zijn. Men kan hierbij bijvoorbeeld denken aan de CONNECTION MACHINE, een parallelle computer met, momenteel, zo'n 64.000 uiterst eenvou-

dige processoren, waarmee men fysische verschijnselen op deeltjesniveau denkt te kunnen onderzoeken. We hebben ons in dit boek maar beperkt tot de behandeling van die systemen waarmee inmiddels enige ervaring is opgedaan en waarmee men denkt voorlopig een heel eind verder te kunnen, in de hoop de lezer een praktisch nuttig inzicht te verschaffen in de huidige mogelijkheden van het parallelle rekenen.

Nog een enkele kanttekening bij het in dit boek gehanteerde taalgebruik. Omdat vrijwel alle literatuur, inclusief de gebruikershandboeken, over vector- en parallelle computers Engelstalig is en bovendien ook de meest gebruikte programmeertaal, FORTRAN, op het Engels georiënteerd is, is er in het betreffende vakgebied een jargon ontstaan dat doorspekt is met Engelse begrippen. Deze begrippen hebben een heel precieze betekenis binnen dit gebied en het zou naar ons idee de duidelijkheid niet bevorderen wanneer deze begrippen overal werden vervangen door, niet algemeen gebruikte, Nederlandse termen. Zouden zelfs ingewijden meteen begrijpen dat met 'koppeling' sluitend het begrip 'chaining' wordt weergegeven? Is elk 'tussengeheugen' een 'cache'? Mede om de lezer de moeite van het terugvertalen te besparen hebben wij ervoor gekozen om algemeen gebruikte Engelstalige termen meestal onvertaald te laten.

EEN

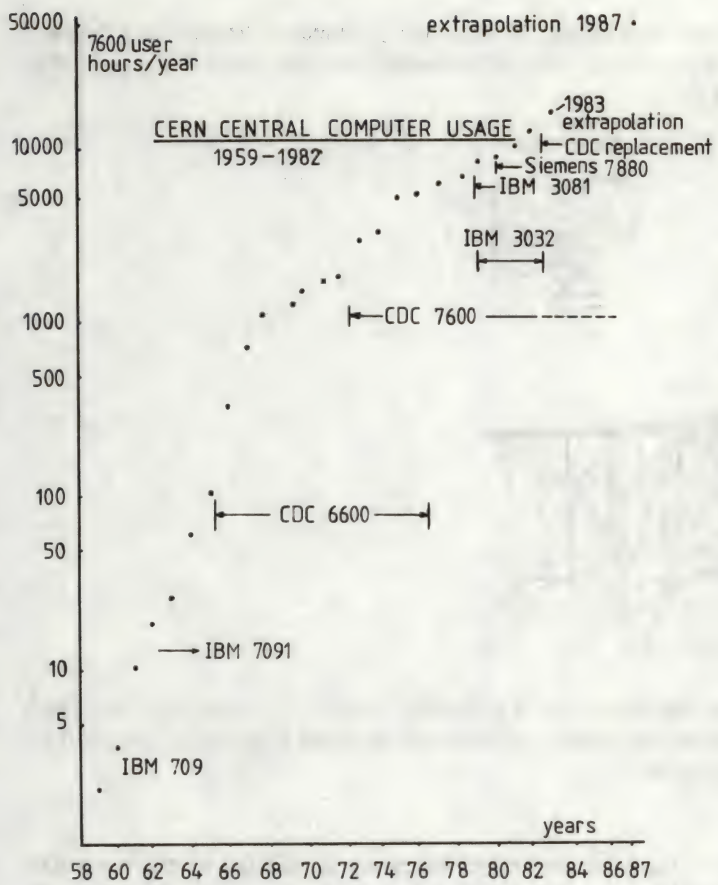
DE BEHOEFTE AAN SNELLE EN GROTE COMPUTERS

Toen programmeerbare computers beschikbaar kwamen werd het daarmee mogelijk om allerlei fysische en chemische verschijnselen, die niet op analytische wijze te beschrijven waren, te bestuderen middels het rekenen aan wiskundige modellen. Dit vergrootte weer de vraag naar snellere en grotere computers, hetgeen op zijn beurt weer aanleiding gaf tot de ontwikkeling van meer verfijnde modellering en aan model en computer aangepaste rekentechnieken. In de huidige tijd woedt deze opstuwende wisselwerking in volle hevigheid voort.

De sterk toenemende behoefte aan computercapaciteit laat zich aardig illustreren met figuur 1, waarin het jaarlijk verbruik aan computertijd bij het kernfysisch laboratorium CERN wordt getoond. Niets wijst erop dat de groei, die uit de grafiek blijkt, in de komende jaren tot stilstand zal komen.

In de toegepaste wetenschappen is de noodzaak tot het werken met modellen evident. Het is bijvoorbeeld niet erg wenselijk, noch praktisch uitvoerbaar, om experimenteel na te gaan wat er gebeurt als de poolkappen smelten (en of dat gebeurt na verhoging van het koolzuurgehalte in de atmosfeer). In de industrie kan de ontwikkeling van nieuwe produkten sterk versneld en verbeterd worden; een goed voorbeeld hiervan is de gang van zaken in de automobielindustrie. Echter ook veel zuiver wetenschappelijk onderzoek is tegenwoordig ondenkbaar zonder computermodellen. Men denke hierbij aan het analyseren en bestuderen van verschijnselen waarvoor experimenten niet of nauwelijks uitvoerbaar zijn, of van grootheden die niet meetbaar zijn (zoals bijvoorbeeld een compleet snelheidsveld bij een stromingsprobleem).

De modellering van een groot aantal verschijnselen is tegenwoordig reeds zo verfijnd, en de rekentechnieken en computers zijn dermate krachtig, dat de verkregen numerieke uitkomsten een bruikbaar beeld geven van de werkelijkheid. Dit maakt kostbare en tijdrovende experimenten grotendeels overbodig. Succesvolle toepassingen van



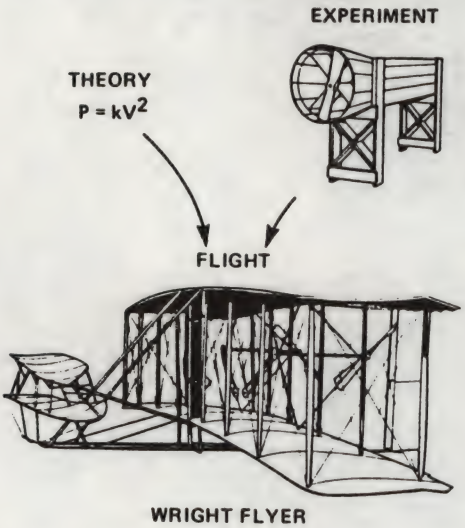
Figuur 1. Het jaarlijks verbruik aan computertijd bij het CERN.

grootschalige numerieke modellering met krachtige computers komt men tegen op uiteenlopende gebieden als luchtvaartaerodynamica, oliereservoironderzoek, kernfusieonderzoek, weersvoorspelling, onderzoek aan moleculaire structuren, waterloopkunde, militair onderzoek (SDI), etcetera.

Als voorbeeld lichten we hier een toepassing op het gebied van de luchtvaart uit. Bij het ontwerpen van een vliegtuig of onderdelen daarvan is het uiteraard zeer gewenst om de stroming rond het object te kennen. Dit is temeer nodig omdat tegenwoordig zeer hoge eisen aan een ontwerp gesteld worden ten aanzien van veiligheid, brandstofgebruik, capaciteit en geluidsreductie. Deze stromingen laten zich beschrijven door de zogenaamde Navier-Stokes vergelijkingen. Voor de genoemde praktische toepassingen kunnen deze vergelijkingen niet analytisch worden opgelost en is men afhankelijk van een zeer gecompliceerde numerieke modellering.

Zoals genoegzaam bekend, is de luchtvaart, in letterlijke zin, reeds van de grond gekomen lang voordat voldoende verfijnde numerieke computersimulaties überhaupt

mogelijk waren. Het eerste vliegtuig (1904) van de gebroeders Wright was gebaseerd op experimenteel onderzoek (o.a. 50 uur windtunnelonderzoek) en sterk vereenvoudigde theorie (zie figuur 2).



Figuur 2. Gemengd experimentele-theoretische aanpak. Overgenomen uit: Eigth International Conference on Numerical Methods in Fluid Dynamics, Proceedings, Aken 1982, Springer-Verlag.

Bij het ontwerpen van moderne verkeersvliegtuigen biedt het windtunnelonderzoek onvoldoende mogelijkheden. Niet alleen is het te tijdrovend (waardoor het moeilijk wordt om allerlei wijzigingen op vleugel- en rompprofielen te bestuderen) en te kostbaar (vanwege de grootte), het komt ook veelvuldig voor dat bepaalde experimenten of metingen in het geheel niet uitvoerbaar zijn.

Een sterk voorbeeld hiervan is de Amerikaanse space shuttle. Bij terugkeer in de dampkring is er een fase waarbij de shuttle zich met circa 20 maal de geluidssnelheid door de (daar nog) ijle dampkring beweegt. Er bestaan op aarde geen installaties waarmee de dan optredende verschijnselen experimenteel bestudeerd kunnen worden. Men was hier dus volledig aangewezen op berekeningen aan een wiskundig model, die dan ook rond 1977 zijn uitgevoerd. Evenwel, bleek veel later, waren deze berekeningen destijds nog dermate onnauwkeurig door de beperkingen die men zich moest opleggen, dat het een haar heeft gescheeld of de eerste shuttle was bij terugkeer in de dampdring verongelukt door een te grote afwijking in de berekende intreehoek. Mede hierdoor is men zich het grote belang van supercomputers en het wetenschappelijk rekenen daarop gaan realiseren (ook van overheidswege).

Bij het (Frans-)Europese space shuttle project (HERMES) denkt men bovendien de ontwikkelingskosten sterk te kunnen reduceren door het verder terugdringen van experimenteel werk door computersimulaties. Men heeft zich daarbij gerealiseerd dat

daarvoor zelfs de momenteel krachtigste supercomputers nog ontoereikend zijn. Men denkt echter dat binnen afzienbare tijd de vereiste capaciteit binnen bereik zal komen.

We besluiten deze inleiding met een uitspraak van dr. Bill Buzbee (Los Alamos, USA), een gerenommeerd expert op het gebied van supercomputergebruik: "Een supercomputer voorzien van adequate software, zoals uitgebreide interactieve grafische faciliteiten, is letterlijk gelijkwaardig met een krachtige telescoop. We kunnen daardoor de verschijnselen binnen een werkende automotor bestuderen, ons een beeld vormen van de lagen in de aardkorst en moleculaire structuren bekijken. Een krachtiger computer heeft dus het effect van een krachtiger telescoop, we kunnen dingen zien die we nog nooit aanschouwden en verschijnselen begrijpen die ons begrip daarvoor te boven gingen." [3].

TWEE

WAT IS EEN SUPERCOMPUTER?

Het is niet goed mogelijk, en ook niet nodig, om een sluitende definitie van het begrip supercomputer te geven. Algemeen wordt deze term gebruikt om er de krachtigste computers mee aan te duiden die er momenteel voorhanden zijn. Met krachtig bedoelen we dan dat deze computers een groot aantal verschillende problemen snel kunnen doorrekenen en dat hun kerngeheugens de enorme hoeveelheden data, inherent aan complexe simulaties, kunnen verwerken.

Men ziet dat deze omschrijving zeer tijdsgebonden is: wat nu nog als supercomputer wordt aangeduid wordt over 10 jaar meewarig als relatief traag ervaren. Men kan, deze definitie hanterend, globaal voor elke tijdsspanne de supercomputers aanwijzen. Zo is, achteraf beschouwd, de CDC 6600 de supercomputer van de jaren '60. Toch wordt deze machine zelden als supercomputer aangeduid, zelfs niet wanneer men zich tot de betrokken periode beperkt.

De term supercomputer kwam eerst in zwang in de tweede helft van de jaren '70 toen door een principiële verandering in de architectuur (de zogenaamde pipelining) de snelheid van de computer opeens met een forse sprong kon toenemen. De eerste commercieel succesvolle computer die volgens deze nieuwe principes werd gebouwd, was de nu al legendarische CRAY-1, spoedig gevolgd door de CYBER 205 van Control Data. De huidige supercomputers, waartoe beide genoemde apparaten nog steeds gerekend worden, wijken allemaal op ingrijpende wijze van hun conventionele voorgangers af, al vertonen ze onderling wel een sterke mate van overeenkomst.

Het feit dat men met de omschrijving supercomputer niet alleen denkt aan een zeer snelle en grote computer, maar ook aan een afwijkende architectuur, wordt nog benadrukt door het volgende. Recentelijk zijn er machines op de markt gekomen die vrijwel volgens dezelfde principes als de huidige supercomputers gebouwd zijn, echter met (veel) tragere componenten en daardoor veel goedkoper. Deze machines, van fabrikanten als Convex, SCS, Gould en Alliant, zijn vaak nauwelijks sneller dan de snelste

conventionele computers. Echter om hun architectuurverschil daarmee aan te geven worden ze vrij algemeen aangeduid als minisupers. De eerste super pc heeft zijn intrede op de markt zelfs gedaan.

We zullen in dit boek geen onderscheid maken tussen supercomputers en minisupers en wat dies meer zij. De beschouwingen en analyses die we presenteren zijn van toepassing op een hele klasse van (super)computers met overeenkomstige elementen in hun architectuur.

Het begrip snelheid is reeds vele malen hier ter sprake gekomen en voordat we nu verder op de specifieke architectuur van supercomputers zullen ingaan, en de consequenties daarvan op de programmering, willen we de lezer eerst een indruk geven van deze snelheid en van de effecten die te bereiken zijn door zorgvuldig rekening te houden met de architectuur.

De huidige supercomputers zien kans om uiteenlopend technisch wetenschappelijk rekenwerk uit te voeren met een gemiddelde snelheid van pakweg 20 miljoen floating point operaties per seconde (een floating point operatie, kortweg flop, is, zeg maar, een optelling of vermenigvuldiging van twee reële getallen). Deze snelheid is over een periode van enkele dagen gemeten bij het Los Alamos Laboratory op een CRAY X-MP die met allerhande soort rekenwerk, karakteristiek voor die werkomgeving, belast werd.

Door geschikte keuzes van rekenschema's en door bij de implementatie daarvan zorgvuldig rekening te houden met de mogelijkheden die de architectuur biedt, kunnen voor sommige, veelvuldig voorkomende, toepassingen zeer veel hogere snelheden behaald worden. Anderzijds is het evenwel ook goed mogelijk dat voor zekere problemen rekensnelheden gehaald worden die niet veel hoger of zelfs lager zijn dan bij de snelste conventionele computers (die snelheden halen van ongeveer 5 miljoen floating point operaties per seconde). Het doen van uitspraken over de kracht van een supercomputer is daarom een hoogst delicate aangelegenheid. Men kan dan ook zeker niet afgaan op de voor elke supercomputer aangegeven topsnelheid, waarover Jack Dongarra (Argonne National Lab.) ooit eens opmerkte: "Topsnelheid is die snelheid welke volgens de fabrikant gegarandeerd niet overschreden kan worden." [17]. Het is verder eigenlijk als met een auto: daarbij is het vrij zinloos om de snelste te kopen als je uitsluitend stadsritjes maakt.

Het betrekkelijke van de snelheden van supercomputers en de potentiële mogelijkheden daarvan wordt nog eens geïllustreerd door de cijfers in tabel 1. Hierin zien we voor een paar supercomputers achtereenvolgens aangegeven: de topsnelheid, de snelheid die behaald werd bij een standaard programma voor een gegeven rekenschema voor het oplossen van een stelsel vergelijkingen, de snelheid van een aangepast programma voor hetzelfde rekenschema (voor een wat groter stelsel) en tenslotte de snelheid voor een aangepast programma voor een aan de architectuur aangepast rekenschema (voor een nog weer wat groter stelsel).

Uit deze tabel zien we duidelijk dat, hoewel de computers op grond van hun topsnelheden nogal verschillen, dit nauwelijks tot uiting komt bij de gemeten snelheden voor een stuk standaardprogrammatuur (vergelijk kolom 1 en 2).

SUPERCOMPUTER	maximum snelheid	standaard software voor een gegeven rekenmethode	aangepaste software voor dezelfde methode	aangepaste software voor een aangepaste rekenmethode
NEC SX-2	1333	43	347	885
CRAY X-MP (1 proc.)	235	39	171	192
NEC SX-1	666	36	224	422
CYBER 205 2-pipe	200	17	31	113
FUJITSU VP-200	533	17	220	422
HITACHI S810/20	630	17	158	
CRAY-2 (1 proc.)	488	15	93	384
CRAY-1S	160	12	76	110

Tabel 1. Snelheden in MFLOPS voor supercomputers onder verschillende omstandigheden (1 MFLOPS =10^6 floating point operaties per seconde)

We zien heel aardig de invloed van de juiste implementatie (kolom 3), waarbij opgemerkt dient te worden dat de snelheidsverschillen voor verschillende implementaties voor een gegeven rekenproces op conventionele computers doorgaans vrij klein zijn (mits men het natuurlijk niet te bont maakt). Tenslotte merken we op dat, vergeleken met de standaardsoftware, dramatische versnellingen bereikt kunnen worden door het kiezen van een rekenschema waarbij de mogelijkheden van de architectuur ten volle kunnen worden uitgebuit en het maken van een zorgvuldige implementatie daarvoor.

Er zij nog opgemerkt dat de gemeten snelheden sterk afhangen van de beschikbare compiler, dat wil zeggen, van de mate waarin de compiler constructies (samengestelde statements) in een (FORTRAN-)programma onderkent, waarvoor optimale code, met betrekking tot de afwijkende architectuur, gegenereerd kan worden.[1]

[1] Het cijfermateriaal in tabel 1 is ontleend aan een publikatie van Dongarra [11].

DRIE

HOGE REKENSNELHEDEN; VECTOR PROCESSING

Tot in de jaren '70 kon men de behoefte aan steeds snellere computers opvangen door verregaande technische hoogstandjes op het beproefde Von Neumann concept. De vacuümbuizen uit de eerste computers werden vervangen door transistoren, daarna kwamen de geïntegreerde circuits en tegenwoordig gebruikt men VLSI technieken. De chips, waarop de circuits zeer dicht zijn opeengepakt, zijn onderling via draadverbindingen op elkaar aangesloten. De rekensnelheden, die voor eenvoudige basisschakelingen tijden in de orde van nanoseconden vergen, maken zeer korte verbindingen noodzakelijk. Immers, een signaal met lichtsnelheid overbrugt in 1 nanoseconde een afstand van 30 centimeter, zodat voor grote computersystemen significante vertragingen kunnen gaan optreden.

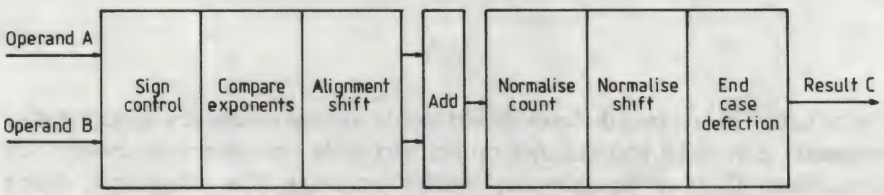
Het nog weer dichter opeenpakken van de chips leidt evenwel tot een onacceptabele (want fatale) hitteontwikkeling, die de fabrikanten van de zeer snelle computers voor grote koelingsproblemen plaatst. De benodigde koelcapaciteit voor, bijvoorbeeld, de CRAY-2 supercomputer wordt verkregen door de chips ondergedompeld te houden in een bad van vloeibaar fluorkoolstof (CF) van zeer lage temperatuur. Dit stelt hoge technische eisen aan het ontwerp. Men moet er onder andere voor zorgen dat een kapot moduul snel vervangen kan worden (binnen een tiental minuten!).

Chips worden momenteel gemaakt op basis van siliciumoxide en men denkt dat er op een termijn van 10 jaar nog een orde aan rekensnelheid gewonnen kan worden op voorwaarde dat men de koelingsproblemen de baas kan. Met behulp van de zogenaamde Josephson Junction techniek en door het gebruik van GaAs (Gallium Arsenide) denkt men dan nog weer behoorlijk aan snelheid te kunnen winnen, maar al bij al zal dat toch onvoldoende blijken om de grote rekensnelheden te halen die men denkt nodig te hebben.

Deze vooruitzichten, alsmede de huidige stand van de techniek, hebben geleid tot computers die qua ontwerp belangrijk afwijken van de inmiddels klassieke Von

Neumann, of seriële, computers. De eerste generatie moderne supercomputers, waaronder de CRAY-1 en de CYBER 205, zijn gebaseerd op het principe van de zogenaamde vectorprocessoren (ook wel pipeline-processoren genoemd). Recentelijk heeft ook de toepassing van meerdere processoren (parallellisme) in commercieel verkrijgbare computers vrij algemeen ingang gevonden. We komen hier nog op terug; we zullen eerst wat dieper op het fenomeen vector processing ingaan.

De klassieke functional unit (binnen een processor) die bijvoorbeeld een optelling[1] van twee floating point getallen uitvoert, doet dat in feite door een aantal achtereenvolgende stappen uit te voeren, zoals: controle op de tekens, vergelijking van de exponenten, het op gelijke hoogte brengen van de exponenten (shift), optellen van de mantissa's en het weer aanpassen van de exponent. Door de functional unit op te splitsen in een aantal specifieke deeleenheden (segmenten), die simultaan hun acties kunnen uitvoeren op verschillende getalparen, kan onder omstandigheden een aanzienlijke versnelling van een rekenproces behaald worden. Voor een schematische voorstelling van een gesegmenteerde optel functional unit, zie figuur 3.

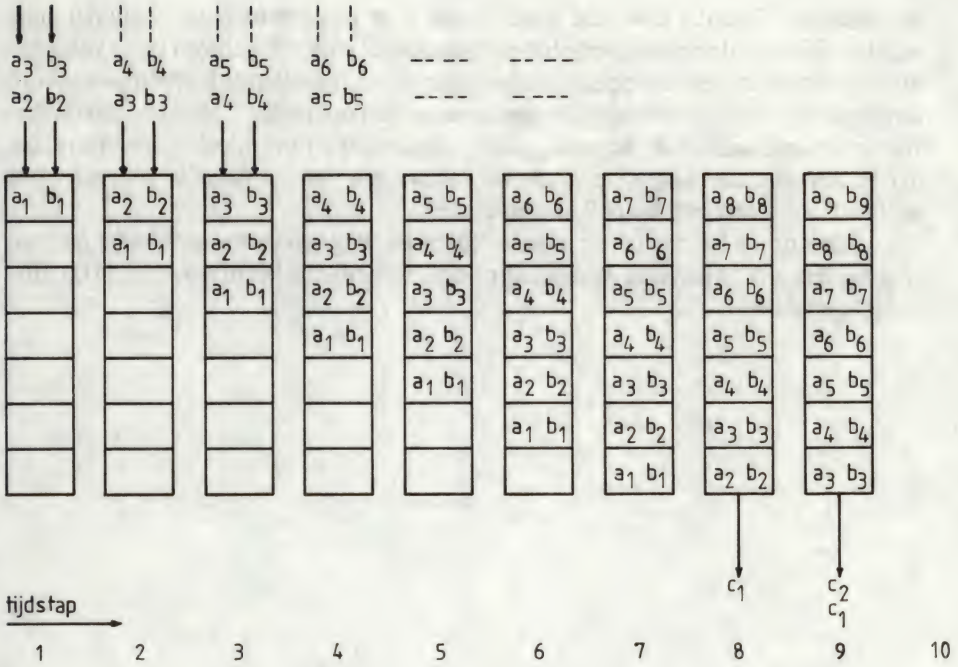


Figuur 3. Gesegmenteerde optel functional unit (ontleend aan Cyber 205)

De deeloperaties, dat wil zeggen: de operatie per segment, zijn zo gekozen dat ze onafhankelijk kunnen worden uitgevoerd en elk 1 *klokcyclus* in beslag nemen. Ook andere functional units, zoals bijvoorbeeld de vermenigvuldig unit en de functional unit die het wegschrijven van getallen naar het geheugen en het ophalen van getallen daaruit regelen, zijn op deze wijze gesegmenteerd. De aantallen segmenten kunnen voor deze functional units verschillend zijn, elk segment vergt wel steeds 1 klokcyclus.

De effectiviteit van een gesegmenteerde functional unit komt pas naar voren als we er voor zorgen dat er ook inderdaad basisoperaties parallel kunnen plaatsvinden. In figuur 4 hebben we schematisch de gang van zaken weergegeven bij het optellen van een groot aantal paren getallen: $c_i = a_i + b_i$, i = 1,2,3,....,N.

[1] We zullen geen onderscheid maken tussen optelling en aftrekking, beide worden door dezelfde functional unit uitgevoerd (a-b=a+(-b))



Figuur 4. Vectorprocessing van $c_i = a_i + b_i$

Met tijdstap hebben we de opeenvolgende klokcycli aangegeven en op elk tijdstip geeft het kolommetje daarboven de toestand weer van een optelunit die uit 7 segmenten bestaat. In elk segment staat aangegeven voor welk paar getallen een deeloperatie wordt uitgevoerd.

In de eerste klokcyclus voert het eerste segment zijn specifieke deelbewerking uit op het eerste getallenpaar a_1, b_1 en schuift daarna het getallenpaar door naar het tweede segment. In de tweede klokcyclus voert het tweede segment zijn deelbewerking uit op het (reeds door segment 1 voorbewerkte) getallenpaar a_1, b_1 en onderwijl neemt het eerste segment alvast het volgende paar a_2, b_2 voor zijn rekening. Net als op een lopende band (pipeline) schuift zo het getalmateriaal de processor binnen, de deelbewerkingen worden uitgevoerd en we zien dat na 7 klokcycli, net zoals bij de klassieke processor, het resultaat c_1 van de optelling $a_1 + b_1$ uit de processor rolt. Daar hebben we weliswaar 7 klokcycli op moeten wachten, maar nu begint het onderwijl onafgebroken bezig zijn van alle deelsegmenten zijn vruchten af te werpen *want elke volgende cyclus* komt er nu een resultaat c_i beschikbaar. Na N+7 klokcycli zijn alle optellingen uitgevoerd en we zien dat er per optelling $(N+7)/N \approx 1$ klokcyclus nodig is geweest (voor N niet al te klein). Omdat er bij de klassieke optelunit 7N klokcycli nodig zouden zijn geweest is er aldus een snelheidswinst met een factor 7 geboekt.

Nu zijn er voor het werkelijk uitvoeren van deze (optel)operaties nog verschillende andere handelingen vereist zoals berekeningen van de adressen waar de getallen in het kerngeheugen te vinden zijn, het ophalen van de getallen en het wegbergen van

de resultaten. Doordat ook deze acties veelal door gesegmenteerde functional units worden uitgevoerd en omdat verschillende functional units vaak als één lange keten aan elkaar kunnen worden gekoppeld (chaining) kan in vele gevallen de snelheidswinst voor de gehele operatie nog belangrijk hoger uitvallen dan een factor 7. Het is ook gebruikelijk dat de optel unit en de vermenigvuldig unit aan elkaar gekoppeld kunnen worden. We komen hier later nog op terug als we zullen laten zien dat dat ook weer een extra snelheidswinst met een factor 2 kan opleveren.

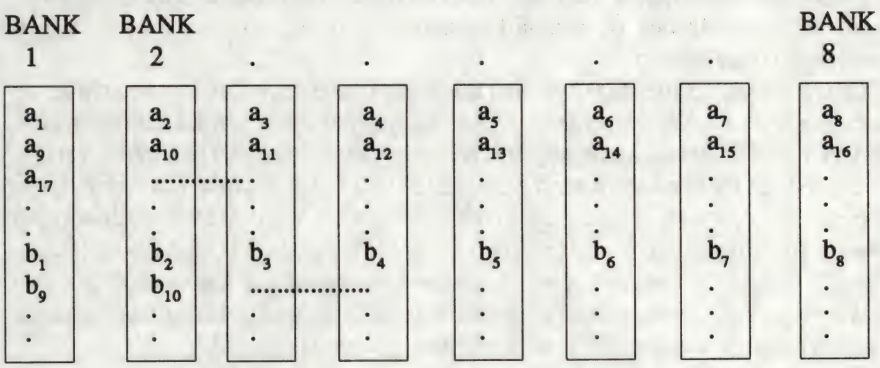
Merk op dat het hier behandelde vector processing een vorm van parallel rekenen is, maar dan wel op een zeer elementair niveau, namelijk op het niveau van de floating point operatie zelf.

VIER

DE ORGANISATIE VAN HET GEHEUGEN

Het is nu wel duidelijk dat het grote voordeel van segmentatie pas tot zijn recht kan komen als we er voor kunnen zorgen dat een gesegmenteerde functional unit ook inderdaad elke klokcyclus een nieuw getallenpaar voorhanden heeft. Dat is technisch lang niet eenvoudig. Het eerste probleem is bijvoorbeeld al dat er maar eens in de zoveel klokcycli (meestal 4) een getal uit een kerngeheugeneenheid kan worden opgehaald, of er naartoe kan worden geschreven. Om dat probleem te ondervangen wordt het geheugen opgedeeld in een aantal verschillende geheugeneenheden. De computer zorgt er nu voor dat van een vector (dat is een rij geïndiceerde variabelen die een getalwaarde hebben) $a_1, a_2, a_3, \ldots, a_N$ de opeenvolgende variabelen in opeenvolgende geheugeneenheden (de zogenaamde *memory banks*) worden opgeborgen.

Op een computer met 8 memory banks, zoals bijvoorbeeld de eerste CRAY-1's, worden dan de waarden van twee vectoren a en b opgeborgen als in figuur 5.



Figuur 5. Onderverdeling van kerngeheugen in banks

Voor het ophalen van de rij a_i, i=1,2,3,.... beginnen we nu met Bank 1 aan te spreken voor a_1. In de tweede klokcyclus kan gestart worden met het ophalen van a_2 uit Bank 2 (bedenk wel dat het ophalen zelf weer een gesegmenteerde operatie is, zodat het ophalen van a_1 nog niet voltooid is als met a_2 begonnen wordt). In de derde klokcyclus kan begonnen worden met het ophalen van a_3, enzovoort. Nadat we Bank 4 hebben benaderd om het ophalen van a_4 te starten, zijn er 4 klokcycli verstreken en kan Bank 1 dus al weer aangesproken worden. Voor het garanderen van een continue stroom a_i naar een functional unit zijn dus in feite 4 memory banks voldoende.

Het aanspreken van een memory bank voordat de vaste memory bank cyclus van (meestal) 4 klokcycli sedert de voorafgaande benadering verstreken is, noemt men een *memory bank conflict*. We moeten dan noodgedwongen wachten totdat de bankcyclus verstreken is en onderwijl kan er dus *niets nieuws* op de lopende band gezet worden en maken de betreffende functional units dus als het ware een paar loze slagen. Stel dat we in het voorgaande voorbeeld, met 8 memory banks, bijvoorbeeld hadden willen (of moeten) rekenen met de getallenrij a_1, a_9, a_{17}, a_{25}, a_{33},..... Deze getallen verblijven allen in Bank 1 en er kan dus slechts eens per 4 klokcycli een nieuw getal bij de reken unit worden aangevoerd. Dat heeft dan vanzelfsprekend een behoorlijk effectiviteitsverlies ten gevolge, want in plaats van iedere klokcyclus kan er nu slechts per 4 klokcycli een resultaat verwacht worden. Memory bank conflicten kunnen, zoals we nog zullen zien, een vervelende rol spelen bij het uitvoeren van allerlei praktische rekenschema's.

We denken nu weer even terug aan het voorbeeld van de optelling in het vorige hoofdstuk: $c_i = a_i + b_i$, i = 1,2,3,....N. Op onze 8 Bank's computer zitten we nu weer middenin de problemen. De waarden a_1 en b_1 moeten terzelfdertijd bij de optel unit arriveren, echter hoe kan dat gerealiseerd worden als, zoals in ons voorbeeld, a_1 en b_1 (en ook alle volgende paren) steeds in dezelfde geheugen Bank voorkomen? Bovendien kan de functional unit die getallen uit het geheugen leest natuurlijk niet tegelijkertijd twee getallen aan de optelunit afleveren.

We hadden natuurlijk wel kunnen voorkomen dat de vectoren a en b hun elementen, in de volgorde zoals we ze nodig hadden, in dezelfde Banks hebben staan. De vector b had bijvoorbeeld opgeborgen kunnen worden door b_1 pas in Bank 5 op te slaan, b_2 in Bank 6 en zo cyclisch verder. Het is snel duidelijk dat dit onze problemen allerminst oplost, wanneer we willen werken met een (groot) aantal verschillende vectoren of wanneer we toevallig ook de getaluitkomsten $a_1 + b_5$, $a_2 + b_6$, $a_3 + b_7$,.... in een rekenschema nodig hebben.

Om de boven geschetste problemen te kunnen omzeilen, heeft men verschillende oplossingen bedacht. We gaan hier ook weer wat dieper op in, omdat kennis hiervan nodig is om zo efficiënt als maar mogelijk is met een supercomputer te kunnen werken.

De door de meeste fabrikanten bewandelde weg is het gebruik van zogenaamde *vectorregisters*, een vorm van geheugen (relatief zeer klein van omvang) die direct door de overige functional units aangesproken kan worden. De regel is nu dat elke (gesegmenteerde) functional unit als operand een heel vectorregister kan accepteren, met daarbij de beperking dat we niet nog eens afzonderlijke elementen uit zo'n vectorregister kunnen halen. We zullen dit weer toelichten met een voorbeeld.

Een vectorregister is, zoals eerder opgemerkt, beperkt van omvang: voor de CRAY-1 beslaat de omvang van 1 vectorregister 64 array–elementen. Omdat de meeste

functional units met 3 vectoren tegelijkertijd te maken hebben (namelijk 2 als ingangsvector en 1 als uitgangsvector) en omdat verschillende functional units aan elkaar gekoppeld kunnen worden waarbij nog weer andere vectoren een rol kunnen spelen, zijn er meerdere vectorregisters (bij de CRAY-1 zijn dat er 8).

Indien we nu twee vectoren a en b elementsgewijs bij elkaar op willen tellen, dan moeten we er eerst voor zorgen dat de elementen van a, dat wil zeggen: a_1, a_2,....,a_N in een vectorregister staan, en hetzelfde geldt voor de elementen van b. Wanneer N kleiner of gelijk is aan 64 dan is dat niet zo'n probleem, echter wanneer N groter is dan 64 dan moet de opteloperatie steeds plaats vinden op porties ter lengte 64 van de betrokken vectoren. Dit onderverdelen in porties van geschikte (vectorregister)lengte wordt door de computer zelf verzorgd. Wij hoeven daar als gebruiker geen rekening mee te houden.

Een bijkomend voordeel van zo'n vectorregister is dat als we de elementen a_1, a_2,.... ook nog voor een andere (vector)operatie willen gebruiken, deze elementen niet opnieuw vanuit het kerngeheugen naar het register *geladen* hoeven te worden, maar dat we gewoon de betrokken processor op het reeds eerder gevulde vectorregister kunnen laten opereren. Gesteld echter dat we nu ook nog over de rij a_2, a_3,.... zouden willen beschikken, dan kunnen we daarvoor niet het betrokken register (dat a_1, a_2, a_3,.... bevat) benutten: een functional unit gebruikt gewoon blindelings de aangeboden vectorregisters als operand en niet bijvoorbeeld het vectorregister vanaf het tweede element. In dit geval moeten we dus gewoon de elementen a_2, a_3,.... opnieuw in een vectorregister laden, zodat de eerste vectorregisterplaats a_2 bevat, de tweede a_3, enzovoort. Overigens bestaat er voor sommige computers wel een aparte 'shift' unit die de elementen van een register een plaats verschoven opbergt in een ander register.

Het vullen van de vectorregisters (de zogenaamde load-operatie) en het wegschrijven van de inhoud naar het geheugen (de zogenaamde store–operatie) wordt voor diverse supercomputers op heel verschillende wijze gerealiseerd. Meestal kunnen deze operaties weer geketend worden aan andere operaties, mits natuurlijk de dan verder benodigde operanden wel voorhanden zijn. Met name kan een vectorregister reeds als operand voor een functional unit optreden onmiddellijk nadat het eerste vectorelement in het register gearriveerd is (dus terwijl het laden van het register nog in volle gang is).

Computers die werken volgens het bovenstaande principe klassificeert men wel als vectorregistercomputers. De computers uit de Cyberfamilie, zoals de CYBER 205 en de ETA 10, kennen dit vectorregisterprincipe niet. Hier worden de vectorelementen in principe rechtstreeks vanuit het geheugen naar de functional units gevoerd en de resultaten worden ook weer rechtstreeks teruggeschreven. Het was daarvoor natuurlijk wel nodig om deze machines met tenminste 2 lees- units (load units) en 1 schrijf-unit (store unit) uit te rusten. Om memory bank conflicten uit te sluiten heeft men variabel in te stellen vertragingen ingebouwd in de stromen van en naar het kerngeheugen. Voor elke vectoroperatie wordt eerst uitgerekend hoe deze vertragingen, eenmalig per vectoroperatie, moeten worden ingesteld en men kan aantonen dat, mits het aantal Banks 16 of groter is, de gehele operatie dan verder ongestoord kan verlopen. Hierbij geldt wel als beperkende voorwaarde dat vectoroperaties alleen kunnen plaatsvinden op opeenvolgende elementen van een rij geïndiceerde variabelen. Dus de rij a_5, a_6, a_7, a_8,...., wordt als *vector* geaccepteerd, daarentegen de rij a_1, a_3, a_5,.... *niet*.

Bij aaneengesloten geïndiceerde vectoren spreekt men van *contiguous vectors*; de

onderlinge *indexafstand* (of *stride*) is dan 1. De meeste vectorregistermachines accepteren ook vectoren met een vaste indexafstand die ongelijk is aan 1, zij het dat dit wel kan leiden tot minder hoge snelheden in verband met eventueel optredende memory bank conflicten.

Het is op deze plaats nog moeilijk te zeggen, welke keuze (hetzij vectorregisters, hetzij direct memory access) het best uitpakt als we hoge snelheden wensen te realiseren. Zoals we zullen zien kan het vullen van de registers een handicap betekenen en ook het in mootjes hakken van langere vectoren werkt in principe vertragend vanwege de logica die voor deze zogenaamde *stripmining* noodzakelijk is. Aan de andere kant vergt het berekenen van de vertragingsfactoren bij direct memory access nogal wat klokcycli, hetgeen leidt tot relatief hoge beginkosten. Deze worden pas weer enigszins teniet gedaan (in relatieve zin) voor vrij lange vectoren.

VIJF

EEN GLOBAAL OVERZICHT VAN DE HUIDIGE SUPERCOMPUTERS

Zoals we reeds zagen, berust het principe van vector processing in feite op een vorm van parallellisme. De klasse van supercomputers omvat veel meer vormen en deze kunnen ruwweg als volgt worden onderverdeeld (Flynn, 1966 [26]):

1. SIMD (= Single Instruction Multiple Data) computers.
   Hierbij zet een enkele instructie vele identieke operaties op meerdere operanden in gang (bijvoorbeeld de optelling van twee vectoren). Tot deze klasse worden de (vector)supercomputers gerekend.

2. MIMD (= Multiple Instruction on Multiple Data) computers.
   Dit zijn de echte parallelle computers, waarbij de verschillende processoren (met elk weer hun eigen functional units) onafhankelijke porties rekenwerk (= meerdere instructies op meerdere data) voor hun rekening kunnen nemen. Binnen deze MIMD-klasse wordt nog weer onderscheid gemaakt tussen de zogenaamde shared memory systemen, waarbij alle processoren direct toegang hebben tot één groot gemeenschappelijk geheugen, en de local memory systemen, waarbij alle processoren slechts hun eigen (lokale) geheugen hebben en waarbij de noodzakelijke uitwisseling van gegevens plaatsvindt door boodschappen (messages) via de onderlinge processorverbindingen. Het is technisch niet mogelijk om bij grotere aantallen processoren deze allen onderling te verbinden. Er wordt daarom voor speciale verbindingsstructuren gekozen en de local memory systemen kunnen dan overeenkomstig deze structuren verder geklassificeerd worden.

Deze onderverdeling is vrij simpel, maar heeft als nadeel dat ze ons vrijwel niets leert dat zou kunnen helpen bij de analyse van de daadwerkelijke verwerking van algoritmen en verder is de onderverdeling nogal ruw. Zoals we nog zullen zien zijn de onderlinge,

voor de analyse relevante, verschillen tussen de computers binnen een categorie zeer groot.

Er zijn wel meer pogingen ondernomen om verschillende computers te abstraheren binnen een enkel model, echter merendeels nog zonder veel succes. Een recente veelbelovende aanpak, die weliswaar duidelijk gecompliceerder is, is voorgesteld in het proefschrift van Wijshoff [64]. Hij laat zien dat men op zinvolle wijze computers kan karakteriseren op grond van een hiërarchisch model voor het geheugen en het vastleggen van de transportmogelijkheden van data. Processoren worden in dit model opgevat als afbeeldingen die aan een geheugenlocatie een (andere) geheugenlocatie toevoegen.

Een voor praktische analyses nuttig model wordt gegeven door Hockney & Jesshope [33]. Zij beschrijven de snelheid van een computer met een gering aantal parameters (2 voor SIMD en 3 voor MIMD [32]. Een nadeel van hun aanpak is dat het model andere parameterwaarden heeft voor verschillende algoritmen en dat het ons a priori weinig inzicht biedt in de mogelijkheden van een gegeven architectuur. Het grote voordeel is echter dat het ons op eenvoudige wijze uitspraken levert over de effectiviteit van een computer. We komen hier in het volgende hoofdstuk nog uitvoerig op terug.

We zullen nu een paar concrete supercomputers de revu laten passeren. Bij elke computer zullen wij in een korte bespreking aangeven op welke karakteristieke wijze hoge rekensnelheden tot stand gebracht kunnen worden.

A. CRAY-1

Dit is de eerste commercieel succesvolle supercomputer geweest, waarvan het eerste model werd afgeleverd in 1976 aan Los Alamos Laboratories. Er zijn over de hele wereld nog tal van CRAY-1's in gebruik, alhoewel deze machine al weer geruime tijd uit produktie is genomen.

De klokcyclus van de CRAY-1 bedraagt 12.5 nanoseconde, zodat er in principe 80 miljoen floating point operaties per seconde (= 80 Mflops) kunnen worden uitgevoerd. Doordat in principe de optel unit en de vermenigvuldig unit aan elkaar gekoppeld ('ge-chained') kunnen worden is een maximum snelheid van 160 Mflops haalbaar.

De CRAY-1 is een vectorregistermachine met 8 vectorregisters ter lengte van 64 elementen elk. Het (gesegmenteerde) transportproces tussen de registers en het geheugen kan slechts plaats vinden met een snelheid van 1 element per klokcyclus (afgezien van opstarttijden) van of naar het geheugen. Zoals bij alle vectorregistermachines draagt het systeem zelf zorg voor het in mootjes hakken, ter lengte van 64, van langere vectoren (de zogeheten stripmining). Deze stripmining geeft natuurlijk wel weer extra overhead, zodat het bepaald onvoordelig is om bijvoorbeeld 2 vectoren ter lengte van 65 elementen bij elkaar op te tellen. Het eerste part van 64 gaat dan met vectorsnelheid en het resterende partje, ter lengte 1, wordt in wezen scalair uitgevoerd (er wordt nog wel wat winst geboekt door het koppelen van acties).

B. CYBER 205

Toen de computerontwerper Seymour Cray de firma Control Data verliet om zelf computers te gaan fabriceren (jawel, de CRAY-1), liet deze firma niet lang op een antwoord

wachten door een concurrerende supercomputer, de CYBER 203, op de markt te brengen (deze werd nog voorafgegaan door de minder succesvolle STAR 100). Enkele minder geslaagde details van het 203-model werden nog weggepoetst en zo kwam in het begin van de jaren '80 het model 205 tot stand.

De CYBER 205 heeft geen vectorregisters, de processoren halen hun data vrijwel rechtstreeks uit het geheugen en schrijven de resultaten ook rechtstreeks terug (direct memory access). Men heeft er dus voor moeten zorgen dat er per klokcyclus 2 getalwaarden uit het geheugen gehaald kunnen worden en 1 resultaat kan worden weggeschreven (zie hoofdstuk 4). Dit alles gebeurt onder controle van de *stream-unit*, die vertragingen in de elementenstromen aanbrengt om gelijktijdige aankomst van de operanden bij een functional unit te bewerkstelligen en om memory bank conflicten (hoofdstuk 4) te vermijden. De consequentie, althans voor de CYBER 205, is wel dat vectoroperaties slechts uitgevoerd kunnen worden op vectoren (of gedeelten daarvan) die bestaan uit elementen met opeenvolgende (oplopende) index (*contiguous vectors*).

De klokcyclus bedraagt 20 ns, zodat de maximum snelheid dientengevolge reeds 50 Mflops zou kunnen bedragen. Dat deze snelheid nog beduidend hoger kan uitvallen is te danken aan een aantal extra voorzieningen.

In sommige gevallen kan de gesegmenteerde vermenigvuldiging weer gekoppeld worden met de gesegmenteerde optelling. Voorwaarde daarbij is dat er bij deze operatie 3 vectoren (immers 'slechts' 3 toegangen per klokcyclus tot het geheugen), 1 scalar en 2 verschillende functional units betrokken zijn: de zogenaamde *linked triad* constructie. Men verwarre dit niet met de chainingsmogelijkheden bij de meeste vectorregistermachines, waarbij simultane operaties van verschillende functional units mogelijk zijn, mits de betrokken (vector)operanden in de registers beschikbaar zijn (of daar juist arriveren). Door de linked triad constructie wordt de maximale snelheid opgekrikt tot 100 Mflops.

De tweede mogelijkheid die de CYBER 205 architectuur biedt is een uitbreiding van het aantal 'vector pipes' (elke pipe bestaat uit een complete set gesegmenteerde functional units, er is echter bijvoorbeeld slechts 1 streamunit voor alle pipes samen) tot 2 of 4. Men spreekt dan van een 1-pipe, een 2-pipe of een 4-pipe model. Het spreekt voor zich dat een dusdanige uitbreiding financiële consequenties heeft. Bij een machine met 2 of 4 vector pipes worden de vectorstromen keurig over de verschillende pipes verdeeld. Zo worden bijvoorbeeld bij de optelling $c_i = a_i + b_i$, $i = 1,...,N$, op een 2-pipe machine de oneven geïndiceerde paren verwerkt op pipe 1 en de even geïndiceerde op pipe 2. Merk wel op dat we nu ook twee keer zoveel data van en naar het geheugen moeten schrijven gedurende elke klokcyclus. De uitbreiding met meerdere pipes heeft een overeenkomstige verhoging van de maximumsnelheid ten gevolge, al moet wel opgemerkt worden dat er steeds langere vectoren vereist zijn om de invloed van de toegenomen start-up tijden relatief weer teniet te doen. Er zijn in de wereld momenteel slechts weinig 1- en 4-pipe machines, het merendeel bestaat uit 2-pipe uitvoeringen. Een van de weinige 1-pipe modellen, die van het Academisch rekencentrum SARA te Amsterdam, is in oktober 1987 opgevoerd tot een 2-pipe computer.

Voor wie dit alles nog onvoldoende soelaas biedt, is er op de CYBER 205 nog de mogelijkheid om in halve precisie te rekenen (de variabelen worden dan in 32 bits gerepresenteerd in plaats van de normale 64 bits; deze 32 bits komen ongeveer overeen

met de standaard precisie van bijvoorbeeld IBM). Het rekenen in halve precisie, hetgeen voor veel wetenschappelijk rekenwerk onvoldoende nauwkeurig is, leidt weer tot nagenoeg een verdubbeling van de snelheid.

Als we alle mogelijkheden bij elkaar nemen, dan kan op een CYBER 205 een maximumsnelheid van 800 Mflops benaderd worden, namelijk bij het uitvoeren van linked triads in halve precisie op een 4-pipe model. We merken tenslotte op dat de invoering van meerdere pipes nog steeds onder het SIMD principe valt, het betreft hier immers het (sneller) uitvoeren van een enkele (vector)instructie.

In het voorjaar van 1988 zijn de eerste exemplaren van de opvolger van de CYBER 205, de zogenaamde ETA10-serie, geplaatst. De ETA10-serie kent verschillende uitvoeringen die zich van elkaar onderscheiden door de lengte van de klokcyclus. Voorts kan een ETA10-machine meerdere processoren omvatten waarmee parallel rekenen ook door Control Data wordt mogelijk gemaakt. Afgezien van de rekensnelheid vertoont de werking van elke ETA10-processor grote overeenkomst met die van een 2-pipe CYBER 205 computer.

C. CRAY X-MP

De vraag naar nog weer snellere computers leidde bij CRAY Research Inc. tot een van de huidige topmodellen, de CRAY X-MP. Ten opzichte van de CRAY-1 werd een verhoging van de snelheid bereikt door een kortere klokcyclus van 9.5 ns, op de meest recente modellen inmiddels weer verder teruggebracht tot 8.5 ns. Dit leidt tot een maximumsnelheid van ca. 235 Mflops, aannemend dat men berekeningen uitvoert waarbij de chaining van optelunit en vermenigvuldigunit benut kan worden.

Omdat verder de communicatie met het kerngeheugen verbeterd werd, zodat het nu mogelijk is om *per klokcyclus* 2 elementen in verschillende registers te laden (dus 2 simultane vectorstromen) en 1 registerelement naar het geheugen weg te schrijven (in totaal dus 3 vectorstromen), kan deze maximumsnelheid ook vaker inderdaad (vrijwel) gerealiseerd worden. Bovendien werden de regels voor chaining wat versoepeld; we zullen op dit aspect niet verder ingaan omdat het voor de gebruiker een geringe rol speelt (de compilerbouwers zagen kans om de FORTRAN compiler daardoor efficiëntere code te later genereren en de invloed van memory bank conflicten wordt er door gereduceerd).

Een wezenlijk verschil met de CRAY-1 (en ook met de CYBER 205) is dat het aantal (job)processoren naar 2 of 4 kan worden uitgebreid. Hierbij is elke jobprocessor in wezen een zelfstandige computer en men moet dit dus niet verwarren met de meervoudige vectorpipe van de CYBER 205. Omdat die 2 of 4 processoren met geheel verschillende klussen (zelfs voor verschillende gebruikers) bezig kunnen zijn hebben we te maken met een computer die als MIMD (shared memory) computer te gebruiken is, waarbij de verschillende processoren SIMD processoren zijn. Deze meervoudige processoren zijn op verschillende manieren te gebruiken. Men kan op een CRAY X-MP/4, dat wil zeggen een 4-processor uitvoering, 4 gebruikers tegelijk laten werken en het apparaat wordt dan in feite gebruikt als 4 afzonderlijke CRAY X-MP/1's. De meeste CRAY X-MP/2 en /4's worden grotendeels op deze wijze gebruikt. Het is evenwel ook mogelijk om *binnen* een programma over meerdere processoren te beschikken, waar-

door, indien er voldoende parallellisme aanwezig is, zo'n programma met een factor van maximaal 4 versneld uitgevoerd kan worden. Deze mogelijkheid wordt gebruikt bij jobs die zeer tijdkritisch zijn, zoals numerieke weersvoorspelling (op een CRAY X-MP/4 te Reading in Engeland). Ook kan men hiermee de zogenaamde turn-around time reduceren (zie verder paragraaf 14.3).

Tenslotte is het nog mogelijk om binnen een programma aan te geven dat een stuk (of stukken) in parallelle vorm, dat wil zeggen onafhankelijk van elkaar, kunnen worden uitgevoerd. Iedere keer als er bij executie van het programma zo'n stuk aan de beurt is, kijkt het systeem of een der overige processoren toevallig niets te doen heeft en als dat het geval is dan wordt zo'n processor ingeschakeld bij de verwerking, zodat de gehele job eerder klaar is.

Als men op een of andere wijze kans ziet om alle 4 de processoren binnen 1 programma op volle snelheid te gebruiken, dan zijn snelheden tot in de buurt van 1 Gflop (= 1 Gigaflop = 1000 miljoen flops per seconde) realiseerbaar. Volgens de definitie van supercomputers zijn daardoor de dagen van de CRAY-1 als zodanig geteld.

De Japanse antwoorden

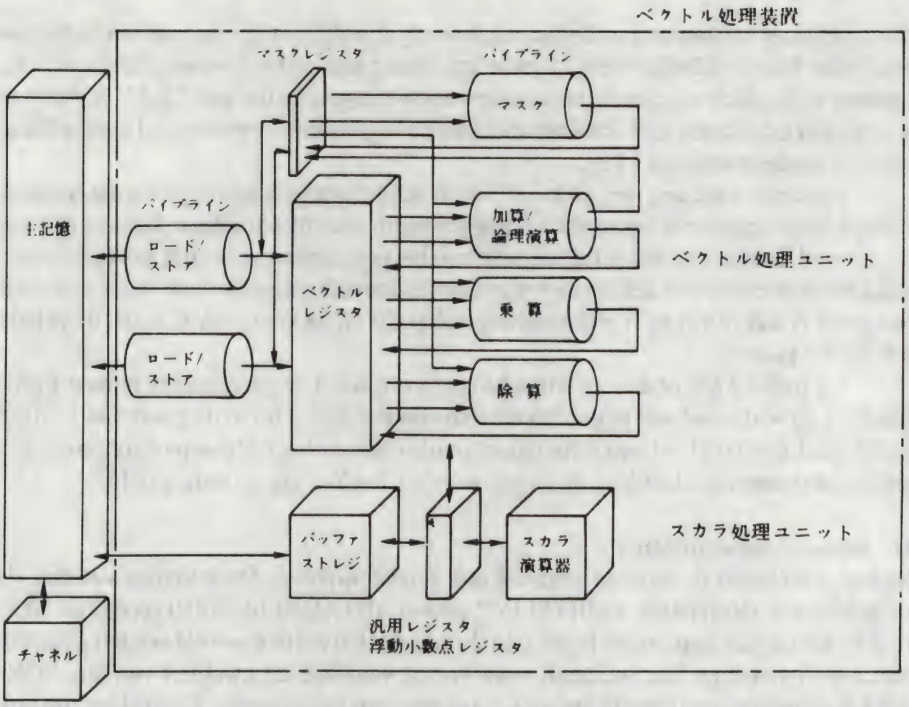
Sedert 1984 levert de Japanse industrie ook supercomputers. Deze komen van drie (!) verschillende fabrikanten: FUJITSI (VP series), HITACHI (de S810 typen) en NEC (SX-typen). Deze computers hebben in de wetenschappelijke wereld opzien gebaard, niet alleen vanwege hun snelheid, maar vooral vanwege de kwaliteit van hun FORTRAN compilers met betrekking tot het automatisch vectoriseren. Sedertdien zijn ook de compilers van CRAY en CYBER verder verbeterd (mede onder druk van de Japanse concurrentie?) zodat men tegenwoordig niet meer van een belangrijk verschil kan spreken. Kenners evenwel ventileren nog regelmatig als hun mening dat met name de FUJITSU VP-compiler als de top beschouwd moet worden.

De architecturen van de Japanse computers bevatten zowel componenten die aan CRAY doen denken als componenten die nauwe verwantschap met de CYBER 205 architectuur vertonen (zie figuur 6 op de volgende pagina).

Voor wat betreft de globale prestatievermogenanalyse voor de meeste lineaire algebra algoritmen zijn er nauwelijks nieuwe elementen aan toegevoegd. Als belangrijkste van deze nieuwe elementen noemen we de mogelijkheid van het combineren van vectorregisters tot langere registers, wat tot gevolg heeft dat de (negatieve) effecten van stripmining, het in mootjes hakken van langere vectoren overeenkomstig de vectorlengte, gereduceerd wordt. Voor niet te lange vectoren kan de hele stripmining logica zelfs uitgeschakeld worden, hetgeen voor vectorlengtes van zeg korter dan 150 tot snelheidsverbeteringen in de orde van 10% kan leiden. Deze mogelijkheden zijn, voor zover ons bekend, aanwezig op FUJITSU en NEC supercomputers. Van de Japanse giganten is de NEC SX-2 momenteel de snelste met een top van 1333 Mflops.

D. NEC SX-2

Het model SX-2 van Nippon Electronic Company is uitgerust met gesegmenteerde functional units met een klokcyclus van 6 ns. Deze klokcyclus zelf leidt reeds tot een

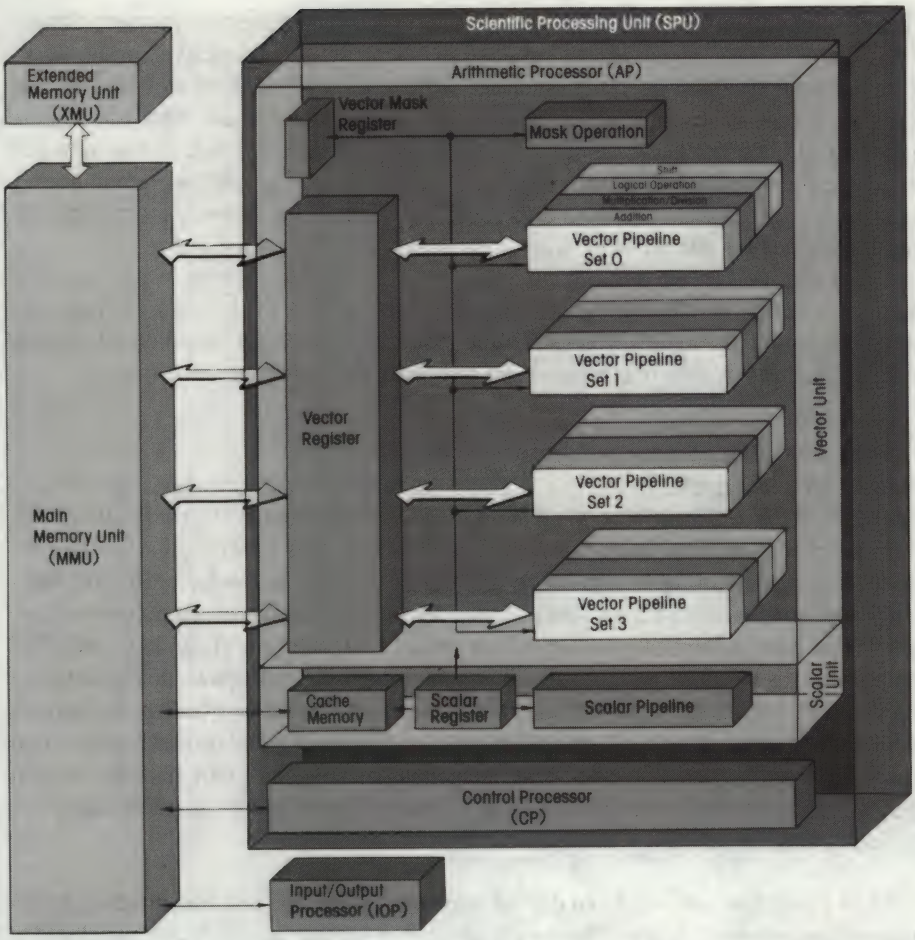


Figuur 6. FUJITSU FACOM VP

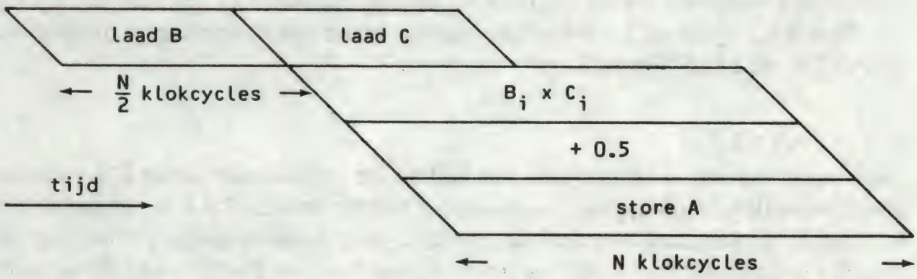
basissnelheid van 167 Mflops. Bovendien is de SX-2 toegerust met 4 vectorpipes, dat wil zeggen dat de vectorstromen worden verspreid over 4 identieke functional units, geheel analoog aan de situatie bij de 4-pipe CYBER 205. Dit verhoogt de topsnelheid tot 667 Mflops. Zie figuur 7 voor een schematisch overzicht van de SX-2 architectuur.

De vector functional units accepteren slechts vectorregisters als operanden. Wanneer de betrokken operanden in de vectorregisters beschikbaar zijn, dan kan weer eventueel de vermenigvuldiging en de optelling van (verschillende) vectorelementen simultaan (of geketend) plaatsvinden. Dit alles tesamen leidt tot de reeds eerder gememoreerde snelheid van maximaal 1333 Mflops.

Het vullen van een vectorregister met elementen uit het geheugen gaat met een snelheid van 2 elementen per klokcyclus, er kunnen evenwel (net als bij de CRAY-1) niet meerdere verschillende vectoren tegelijk geladen worden. Dit betekent dat het laden van een vectorregister circa de helft van de tijd kost die nodig is voor het vermenigvuldigen van haar inhoud met een ander register. Het transport van een register naar het geheugen (de store operatie) kan plaatsvinden met de snelheid van 1 element per klokcyclus. Indien een algoritme dat toelaat, dan kunnen verschillende operaties zoals een load, een store en een vectorvermenigvuldiging simultaan (dat wil zeggen: geketend) plaatsvinden (zie figuur 8).



Figuur 7. NEC SX-2 architectuur.



Figuur 8. Schematisch verloop van de berekening van $a_i = b_i * c_i + 0.5$, $i = 1,2,....,N$. Met de schuine zijden wordt de opstarttijd voor de betreffende processor aangeduid.

Vanzelfsprekend is niet al het rekenwerk geschikt voor verwerking via vectoroperaties. Het is daarom voor veel toepassingen van groot belang dat ook het niet-vectoriële werk, de zogenaamde scalaire operaties, snel kan worden uitgevoerd. Men ziet dit direct in door zich te realiseren dat wanneer 1/3 deel van de operaties van een algoritme scalair werk betreft, men het geheel niet sneller krijgt dan een factor 3 ook al zou men het vectoriële gedeelte oneindig snel uitvoeren. Door ook op scalair niveau segmentatie en chaining toe te passen is de SX-2 ook voor overwegend scalair werk momenteel de snelste machine ter wereld.

Tot slot merken we nog op dat de bank cycle time (van belang in verband met memory bank conflicten) 13 klokcycli bedraagt (voor de CRAY-1 is dat 4). Dat betekent dat een algoritme dat al de vectorelementen uit dezelfde memory bank betrekt (er zijn 128-256 memory banks) mag rekenen op een forse vertraging.

E. IBM 3090 VF

Tegen het eind van 1985 kondigde IBM de mogelijkheid aan om haar topmodellen 3090-200, 3090-400 en 3090-600 uit te breiden met zogenaamde Vector Facility (VF) units: het model 200 met maximaal 2 units, het model 400 met maximaal 4 units en het model 600 met maximaal 6 units. Deze vectorunits werken met vectorregisters en hebben een klokcyclus van 18.5 ns. Aangezien geheel analoog aan de overige supercomputers de optelling en vermenigvuldiging van vector operanden simultaan kan plaatsvinden, biedt elke VF-unit een topsnelheid van maximaal 108 Mflops en de theoretische top van het model IBM 3090-400 met 4 VF's zou dus 432 Mflops bedragen. Om een voldoende snelle toevoer van data naar de functional units mogelijk te maken heeft men tussen de vectorregisters en het centraal geheugen nog een apart tussengeheugen (cache) geplaatst. Dit is van grote invloed bij het realiseren van hoge rekensnelheden (zie ook paragraaf 6.3).

Het kon natuurlijk niet uitblijven dat het segmentatieprincipe ook met goedkopere (= tragere) componenten zou worden toegepast om een verhoging van de rekensnelheid te bewerkstelligen. Dit leidde dan ook tot het ontstaan van de zogenaamde supermini's, smalend ook wel eens Crayettes genoemd. Dit zijn, populair gesproken, computersystemen met een prestatievermogen van hooguit enkele tientallen Mflops, echter voor een prijs die een fractie van die van een echte supercomputer bedraagt, zeg maar de prijs van een flink VAX-systeem. De bekendste representanten zijn momenteel afkomstig van CONVEX, ALLIANT en SCS.

F. CONVEX C-1

Deze machine heeft een klokcyclus van 100 ns. De architectuur van de C-1 vertoont, althans voor de gebruiker, grote overeenkomst met die van de CRAY-1, met name voor wat betreft het vectorregistermechanisme, load–store mogelijkheden en chaining. De oplettende lezer leidt hieruit eenvoudig een topsnelheid van 20 Mflops af. Er wordt de gebruiker ook nog de mogelijkheid geboden te rekenen met (IBM-)single precision variabelen, hetgeen een verdubbeling van de rekensnelheid tot gevolg kan hebben (40 Mflops).

De firma CONVEX gaat met name prat op de vectoriserende kwaliteiten van haar FORTRAN compiler. Op zeker moment zag haar FORTRAN f77 compiler kans om 78% van de loops uit de zogenaamde Orszag-Mendez collectie te vectoriseren, tegen 69% voor de FUJITSU compiler en 57% voor de CRAY-1 compiler. Bedenk hierbij wel dat dit een tussenstand is, de prestaties van alle compilers gaan nog steeds vooruit. Bovendien zegt dit gegeven slechts dat bepaalde loops automatisch gevectoriseerd werden; we worden uit die percentages allerminst gewaar hoe efficiënt die vectorcode dan wel was.

G. ALLIANT FX

Deze computer is gebaseerd op een processoreenheid (CE) met gesegmenteerde functional units en vectorregisters (ter lengte van 32 woorden (64 bits elk)). De klokcyclus bedraagt 170 ns. Deze computer kent alleen chaining voor de zogenaamde 'linked triad' constructie (bijvoorbeeld $x_i = x_i + a * y_i$, zie de bespreking van de CYBER 205). Hierdoor kan een maximumsnelheid van circa 11.8 Mflops bereikt worden (in met CRAY en CYBER vergelijkbare precisie).

Een verdere verhoging van de rekensnelheid wordt gehaald door meerdere processoren (tegen meerprijs natuurlijk) in te schakelen, tot een maximum van 8. Deze processoren kunnen onafhankelijk opereren en hebben allen toegang tot hetzelfde (shared) memory. Parallelle verwerking vindt automatisch plaats op do-loop niveau, dat wil zeggen dat verschillende doorgangen van een loop (verschillende loop-index waarden), die ook ingewikkelde scalaire constructies mag omvatten, kunnen plaatsvinden op verschillende processoren. Deze loop distributie wordt door de FORTRAN compiler verzorgd.

(Mini)supercomputergebruik in Nederland

Tot slot geven we een kort overzicht van de stand van zaken aan het supercomputerfront in Nederland. Voor zover ons bekend op dit moment, november 1987, is de situatie als volgt:

a. Via het ENR te Petten kan men toegang verkrijgen tot een CRAY X–MP/2 van het CRAY Research Centre te Bracknell, Engeland. Toewijzing van rekenfaciliteiten en de financiën daarvoor wordt geregeld door de Werkgroep Gebruik Supercomputers (WGS).
b. Het KNMI maakt voor een groot deel van haar wetenschappelijk rekenwerk, alsmede voor de samenstelling van de dagelijkse weerberichten, gebruik van de CRAY X-MP/4 computer die staat opgesteld bij het European Centre for Medium range Weather Forecast te Reading (Engeland). Bovendien heeft het KNMI voor eigen ontwikkelingen een CONVEX C-1 in huis.
c. Het universitaire rekencentrum SARA te Amsterdam beschikt over een CYBER 205 2-pipe model. Ook hier kan de toewijzing van rekenfaciliteiten plaatsvinden via de WGS.
d. Het KSEPL (Shell Research) te Rijswijk heeft een eigen CRAY-1. Tot voor kort kon men ook via het ENR, eventueel met tussenkomst van het WGS, toegang tot deze

CRAY-1 krijgen. Sedert het gereedkomen van de onder a genoemde verbinding is deze mogelijkheid komen te vervallen.

e. Philips Eindhoven heeft eind 1987 haar IBM 3090 uitgebreid met VF-units.
f. Het NLR (Nederlands Lucht en Ruimtevaart Laboratorium) heeft sinds november 1987 een NEC SX-2 computer voor het uitvoeren van groot rekenwerk in verband met numerieke storingsmodellen. Het betreft hier pas de tweede NEC SX-2 buiten Japan en het is de eerste supercomputer ter wereld beneden de zeespiegel.
Naar verwachting zal ook hier een deel van de rekencapaciteit, door bemiddeling van de WGS, ten goede van de Nederlandse academische gemeenschap kunnen komen.
g. Er staan inmiddels meerdere CONVEX C-1 computers opgesteld in Nederland, waaronder twee bij het rekencentrum van de TU-Delft.
h. De sterrenwacht te Dwingelo, alsmede de TU-Eindhoven, zijn de eerste instellingen die over ALLIANT FX computers beschikken.

Eind 1986 stonden er in West-Europa 53 (echte) supercomputers opgesteld. We geven de verschillende typen met tussen haakjes de aantallen (ontleend aan [22]): CRAY-1 (13), CRAY X-MP (22), CRAY-2 (3), CYBER 205 (7), FUJITSU VP (10) en IBM 3090 + VF (23).

Dit totale aantal steekt schril af bij de toekomstvisie van een bekend Amerikaans expert die rond 1975 voorzag dat er in de toekomst hooguit behoefte zou zijn aan 2 à 3 CRAY-1's voor de gehele wereld. Alleen al bij Los Alamos Lab's staan er nu een stuk of 7 supercomputers. Het is natuurlijk gemakkelijk achteraf de spot te drijven met de ongelukkige die toen tenminste een visie had; het geeft echter wel aan dat de wisselwerking tussen numerieke modellering en rekencapaciteit een veel hogere vlucht heeft genomen dan men ooit heeft durven dromen.

Als men geïnteresseerd is in nieuwtjes en nieuwe ontwikkelingen rond supercomputers dan kan men veel van zijn gading vinden in het blad *Supercomputer* (red. A. Emmen, J. Hollenberg, R. Llurba en A. van der Steen).

ZES

EFFECT VAN VECTORISATIE OP DE REKENSNELHEID

In dit hoofdstuk zullen we proberen een indruk te krijgen van het rendement van gevectoriseerd (of fijnschalig parallel) uitvoeren van een gedeelte van een rekenschema. We hebben reeds een indicatie gehad dat men geen te overspannen verwachtingen mag koesteren; immers indien 90% van alle bewerkingen tijdens uitvoering van een programma als vectoroperaties worden uitgevoerd, dan blijft er, zelfs al zouden we de tijd voor die 90% tot een verwaarloosbare hoeveelheid weten terug te dringen, nog een traag gedeelte over ter grootte van 10% van de oorspronkelijke klus. Dientengevolge kan de snelheidswinst nooit meer dan een factor 10 bedragen (en dat is natuurlijk ook nooit weg).

6.1 Het effect van de scalaire snelheid

Het nadelig effect van de niet met vectorsnelheid uitgevoerde operaties in een algoritme op de totale snelheid wordt beschreven door de wet van Amdahl [1], die we hier zullen afleiden en van commentaar voorzien.

Het is gebruikelijk om de snelheid van een supercomputer voor een gegeven serie bewerkingen (floating point operaties = flop) uit te drukken in de eenheid Mflops (ook wel genoteerd als Mflop/s), dat is in eenheden van een miljoen flops per seconde. Als dus N flops uitgevoerd worden in t microseconde (μsec = 10^{-6} sec), dan is de snelheid s = N/t Mflops, en omgekeerd als een computer N flops uitvoert met een (gemiddelde) snelheid van s Mflops, dan is de verwerkingstijd t = N/s μsec.

Laten we nu eens aannemen dat een ons interesserend algoritme bij daadwerkelijke uitvoering N flops vergt en dat een fractie p hiervan, dus pN flops, uitgevoerd wordt

met vectorsnelheid v Mflops. Het overige gedeelte, N - pN = (1-p)N flops wordt met scalaire snelheid, dat wil zeggen niet via vectoroperaties, uitgevoerd, aangegeven met s Mflops, waarbij in het algemeen s veel kleiner zal zijn dan v.

Voor de totale verwerkingstijd geldt dientengevolge

$$t = \frac{pN}{v} + \frac{(1-p)N}{s} = N(\frac{p}{v} + \frac{1-p}{s})\ \mu\text{sec}.$$

Hoe we ook ons best doen met het opvoeren van de (vector)snelheid v voor het gedeelte p, de totale tijd is altijd groter dan N (1-p)/s μsec. Dat betekent dat de reductie in rekentijd, tengevolge van het rekenen met vectorprocessoren, dus altijd beperkt blijft tot een factor 1-p ten opzichte van het scalair uitvoeren van het gehele algoritme. Als p dus niet zeer dicht bij 1 ligt dan heeft het weinig zin om, althans voor het gegeven algoritme, de vectorsnelheid steeds verder op te voeren.

Voor de snelheid R_A voor het hele algoritme krijgen we (volgens de definitie het aantal flops gedeeld door de tijd in μsec):

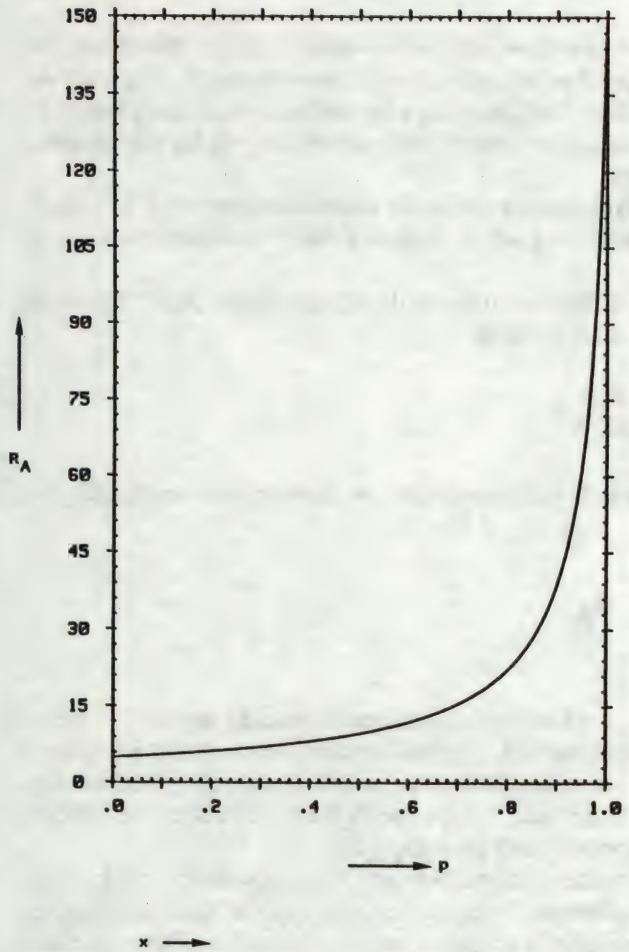
$$R_A = \frac{1}{\frac{p}{v} + \frac{1-p}{s}}$$

Dit is een eenvoudige vorm van de *wet van Amdahl*. We zien hieruit de beperkingen ten gevolge van de scalaire snelheid, er geldt namelijk altijd $R_A < s/(1-p)$ en voor kleine p kan de uiteindelijke snelheid dus nooit erg indrukwekkend zijn.

In figuur 9 op de volgende pagina zien we de snelheid R_A voor het geval dat de vectorsnelheid 150 Mflops en de scalaire snelheid 5 Mflops bedraagt (deze getallen zijn representatief voor een situatie op de CRAY-1 [21]). We zien hier op grafische wijze dat het percentage van het programma dat vectoriseerbaar is dicht bij de 100% moet liggen om een significante verbetering van de rekensnelheid te krijgen ten opzichte van de scalaire snelheid s. Het is dus van wezenlijk belang om, naast een hoge vectorsnelheid, er *ook* voor te zorgen dat de scalaire snelheid zo hoog mogelijk is.

6.2 De invloed van de vectorlengte op de vectorsnelheid

We hebben wel al ter sprake gebracht dat het bij vectorprocessing even kan duren voordat de eerste resultaten van de lopende band rollen, maar we hebben vervolgens onze aandacht uitsluitend gericht op de omstandigheid dat er daarna elke klokcyclus een volgend resultaat komt. Nu zal geen autofabrikant op het idee komen om een lopende band in elkaar te knutselen als er slechts een paar modellen moeten worden gebouwd. Evenzo mag men natuurlijk geen hoge verwachtingen koesteren van de rekensnelheid als men vectorprocessoren gaat gebruiken voor de verwerking van vectoren van bijvoorbeeld



Figuur 9. De wet van Amdahl voor s = 5 en v = 150

lengte N=2. De vectorlengte N zal wel wat groter moeten zijn voordat het segmentatie-effect echt goed tot zijn recht komt. Die afhankelijkheid van de snelheid van N zullen we in deze paragraaf nader gaan beschouwen.

We bekijken een rekenschema A, dat bestaat uit het uitvoeren van een aantal vectoroperaties op vectoren van gelijke lengte N. Het totale aantal flops dat uitgevoerd wordt zullen we aanduiden met $k_A N$. De parameter k_A geeft dus het aantal flops per loop aan.

Bij verwerking op een vectorcomputer worden er een aantal vectorpipelines (lopende bandjes) in werking gezet. Wellicht kunnen er een paar achter elkaar geschakeld worden (bijvoorbeeld via chaining), andere pipelines zullen misschien moeten wachten op de afhandeling van vorige pipelines (je kan bijvoorbeeld niet 2 pipelines opzetten die beide de optel unit gebruiken als er slechts 1 optel unit is). Hoeveel verschillende pipe-

lines er totaal in werking gezet moeten worden weten we dus nog niet (dat zal later pas aan de orde komen), maar voor het gemak stellen we dat aantal maar op k_c. Afgezien van opstarttijden komt er per pipeline elke klokcyclus een resultaat beschikbaar. Indien we de totale opstarttijd noteren als u, dan zijn er voor de verwerking van het rekenschema A dus $u + k_c N$ klokcycli nodig.

De lengte van de klokcyclus geven we aan in microseconden met c. Het aantal klokcycli per µsec is dan 1/c en dit geeft de basissnelheid P van de machine aan in Mflops: P = 1/c.

Voor het rekenschema A, berekenen we nu de Mflops-waarde R(A,N) door het aantal bewerkingen per µsec uit te rekenen:

$$R(A,N) = \frac{k_A N}{(u+k_c N)c} = \frac{k_A N}{u+k_c N} P$$

Voor steeds grotere waarden van N wordt steeds beter de asymptotische snelheid $R_\infty(A)$ benaderd:

$$R_\infty(A) = \lim_{N\to\infty} R(A,N) = \frac{k_A}{k_c} P.$$

De verhouding tussen k_A en k_c geeft dus in feite aan hoe effectief de snelheid (in wezen de lengte van de klokcyclus) benut kan worden. De maximumsnelheid wordt eerst bereikt, zoals we al zagen, als er per klokcyclus zowel een optelling als een vermenigvuldiging kan worden uitgevoerd, 2 flops dus. In dat geval is de maximumsnelheid dus 2P, in alle andere gevallen geldt dat $k_A/k_c \leqq 2$.

Al is de asyptotische snelheid ons nu bekend (vooropgesteld dat we k_A en k_c kennen), het is zeker zo interessant om te weten voor welke waarden van N een gegeven machine al zo'n beetje op snelheid is gekomen, zeg bijvoorbeeld op halve snelheid $\frac{1}{2}R_\infty(A)$. De waarde waarvoor algoritme A de helft van zijn maximale snelheid behaalt wordt aangegeven met $n_{\frac{1}{2}}$. Voor deze waarde moet dus gelden:

$$\frac{k_A n_{\frac{1}{2}}}{u+k_c n_{\frac{1}{2}}} = \frac{1}{2}\frac{k_A}{k_c}$$

en hieruit volgt $u = k_c n_{\frac{1}{2}}$, dus

$$n_{\frac{1}{2}} = u/k_c$$

Dit betekent dus dat de machine op halve snelheid komt, althans voor rekenschema A, zodra de vectorlengte gelijk wordt aan het quotiënt van de totale opstarttijd (gemeten in aantallen klokcycli) en het aantal klokcycli nodig per doorgang van de loop (k_c). Een kleine waarde voor $n_{\frac{1}{2}}$ is zeer welkom, want dat betekent dat de machine al spoedig op een redelijke snelheid ligt.

De verwerkingstijd $t_A(N)$ voor algoritme A, uitgedrukt in μsec, kunnen we schrijven als

$$t_A(N) = (u+k_cN)c = \frac{k_c n_{\frac{1}{2}} + k_c N}{P} = \frac{k_A}{R_\infty(A)}(N+n_{\frac{1}{2}})$$

en voor de snelheid in Mflops geldt uiteraard

$$R(A,N) = \frac{k_A N}{(u+k_cN)c} = \frac{k_A NP}{(n_{\frac{1}{2}}+N)k_c} = R_\infty(A)\frac{1}{n_{\frac{1}{2}}/N+1} .$$

Uit deze formule ziet men dat voor $N = qn_{\frac{1}{2}}$ de rekensnelheid

$$\frac{q}{q+1} R_\infty(A)$$

bedraagt.

De bovenstaande analyse is geïnspireerd op werk van Hockney & Jesshope [33].

Voor vectorregistermachines kunnen de kosten van 'stripmining' (het in moten hakken van de vectoren) ook nog verwerkt worden in dit model door de waarde van k_c aan te passen. We geven de vectorregisterlengte aan met r. Er vindt dus [(N-1)/r] keren overhead plaats, waarbij we met [x] het naar beneden afgeronde resultaat van de waarde van x aanduiden. Indien de totale stripmining overhead voor rekenschema A per vectormoot p klokcycli bedraagt dan wordt de rekentijd dus

$$t_A(N) = (u + [N-1/r]p + k_cN)c \approx (u + (N/r)p + k_cN)c \quad \text{(voor grote N)}$$

$$= (u + \tilde{k}_cN)c \quad , \quad \text{met } \tilde{k}_c = k_c + p/r.$$

De formule voor R(A,N) wijzigt net zo, met $R_\infty(A) = (k_A/\tilde{k}_c)P$

Resumerend zien we dat de rekensnelheid voor een algoritme A wordt gekarakteriseerd door de parameters $R_\infty(A)$, k_A en $n_{\frac{1}{2}}$. De waarde van $R_\infty(A)$ wordt simpelweg bepaald door de klokcycluslengte c en de verhouding tussen aantallen flops en cycli.

De waarde van $n_{\frac{1}{2}}$ geeft de waarde van de vectorlengte aan waarvoor de machine de helft van de asyptotische- of limietsnelheid bereikt. Uit de formule voor R(A,N) zien we ook hoe dat voor andere waarden van N ligt. Als bijvoorbeeld $N = 10n_{\frac{1}{2}}$ dan is de snelheid 90% van de limietsnelheid. Een hoge opstarttijd (= hoge $n_{\frac{1}{2}}$) geeft dus aanleiding tot matige snelheden als de vectorlengtes bescheiden zijn. Voor samengestelde algoritmen met verschillende vectorlengtes kan men desgewenst de snelheid analyseren door gebruik te maken van de wet van Amdahl (vorige sectie) en de hier gegeven uitdrukkingen voor $t_A(N)$ en R(A,N). We zullen daar nog voorbeelden van tegenkomen.

6.3 (R_∞ ,$n_{\frac{1}{2}}$) - Metingen

Voordat we ons zullen verdiepen in de vraag waarom bepaalde rekenconstructies geschikter zijn voor een vectorcomputer dan andere en hoe we daarvan kunnen profiteren, presenteren we eerst rekensnelheden voor eenvoudige DO-loops om een gevoel voor rekensnelheden te krijgen. Deze rekensnelheden zullen gepresenteerd worden door de waarden R_∞ (de rekensnelheid voor een zeer hoge vectorlengte) en $n_{\frac{1}{2}}$ (de vectorlengte waarbij de snelheid $\frac{1}{2}R_\infty$ bedraagt). We zullen eerst toelichten hoe deze parameters bepaald kunnen worden.

Op alle computers kan men opvragen hoeveel CPU-tijd er op zeker punt in de berekeningen is verbruikt. Hiermee kan de CPU-tijd voor een DO-loop van gegeven lengte gemeten worden door het verschil te nemen van de vlak voor en de vlak na de loop gemeten CPU-tijden. Meestal is de tijdmeting niet vreselijk nauwkeurig (eerder vreselijk onnauwkeurig), dat wil zeggen dat de fout in de meting voor een korte loop vaak in de orde van de tijd voor de loop zelf is. Om tot een nauwkeuriger resultaat te komen voeren we de loop een (groot) aantal keren uit, bijvoorbeeld:

```
                    NREP = 500
                    CALL CPUTIME(TIJD1)
                    DO 100 IREP = 1,NREP
de te                 DO 10 I = 1,N
meten                   X(I) = A(I)*B(I)+C(I)
loop          10      CONTINUE
              100   CONTINUE
                    CALL CPUTIME(TIJD2)
                    TIJD = (TIJD2-TIJD1)/NREP
```

Dit ziet er tamelijk voor de hand liggend uit, echter in de praktijk leidt het vrij vaak tot zeer misleidende resultaten. We zullen de twee belangrijkste valkuilen bespreken.

1. Veel compilers zijn tegenwoordig zo slim dat ze detecteren dat er in de herhaald uitgevoerde loop steeds hetzelfde gebeurt. In zo'n geval wordt wel de waarde van IREP verhoogd, maar de loop wordt slechts één keer uitgevoerd en dat leidt tot verrassend hoge Mflops waarden (indien we er vanuit gingen dat er NREP keren iets zou gebeuren). We moeten dus verhinderen dat de compiler kan detecteren dat er invarianten in de binnenste loop zitten. De standaard manier om zoiets te doen is een routine aan te roepen:

```
                    NREP = 500
                    CALL CPUTIME(TIJD1)
                    DO 100 IREP = 1,NREP
                      DO 10 I = 1,N
                        X(I) = A(I)*B(I)+C(I)
              10      CONTINUE
```

```
        CALL DUMMY
100   CONTINUE
      CALL CPUTIME(TIJD2)
      TIJD = (TIJD2-TIJD1)/NREP
```

en aan het programma wordt toegevoegd:

```
      SUBROUTINE DUMMY
      RETURN
      END
```

De meeste compilers zien niet dat DUMMY ook niet veel doet, ze gaan er domweg vanuit dat er in de subroutine aan de waarden van de betrokken vectorelementen kan zijn gesleuteld en voeren daarom de te meten loop elke keer braaf opnieuw uit. Door het toevoegen van de aanroep van DUMMY is de DO-loop wat duurder geworden en we moeten daarvoor corrigeren, bijvoorbeeld:

```
      NREP = 500
      CALL CPUTIME(TIJDA)
      DO 1 I = 1,NREP
        CALL DUMMY
1     CONTINUE
      CALL CPUTIME(TIJDB)
      TIJDCOR = TIJDB-TIJDA
```

Bij het meten van de tijd voor de 'echte' loop, vervangen we dan de berekening van TIJD door:

```
      TIJD = (TIJD2-TIJD1-TIJDCOR)/NREP
```

Een enkele keer gebeurt het dat een compiler alsnog detecteert dat DUMMY niets doet (de CONVEX C-1 FORTRAN compiler van februari 1988) en alsnog een (in dit geval) ongewenste optimalisatie uitvoert. In zo'n geval kan men DUMMY bijvoorbeeld apart compileren, of binnen DUMMY iets aan de array's veranderen.

2. Sommige vectorcomputers realiseren hun hoge rekensnelheden door tussen de zeer snelle vectorregisters en het (relatief trage) centraal geheugen een klein (snel) tussengeheugen (cache) te plaatsen (meestal slechts enkele duizenden woorden groot). Voorbeelden hiervan zijn de CONVEX C-1, ALLIANT FX/8 en de IBM 3090 met VF. Voordat een vectoroperatie kan worden uitgevoerd moeten de data eerst naar het cachegeheugen getransporteerd worden en dat heeft meestal een vertragend effect. Als daarna een loop een aantal keren wordt uitgevoerd, dan kan verversing van het cachegeheugen achterwege blijven. Pas zodra er geheel nieuwe arraygedeelten (dus andere indexwaarden) worden aangesproken vindt aanpassing via het geheugen plaats. In principe zien de functional units alleen het cache-

geheugen als feitelijke geheugen, het echte centraal geheugen wordt gezien als een virtueel geheugen. Er treden daarom bij een cachegeheugen dezelfde effecten op als bij een virtueel geheugen (of in het algemeen: hiërarchisch geheugen) en deze effecten zullen worden toegelicht in paragraaf 10. Een en ander heeft tot gevolg dat het vertragend effect van het cachegeheugen, voor niet al te lange loops (waarbij de betrokken vectoren allen geheel in het cachegeheugen passen), in onze tijdmeting slechts 1 keer, in plaats van NREP keren, optreedt. We moeten er dus voor zorgen dat, indien we het cacheeffect wensen mee te nemen, er geforceerde cacheverversing optreedt. Dat kan men doen door in DUMMY bijvoorbeeld een lange vector te kopiëren. We zullen hier nog op terug komen bij de presentatie van de meetwaarden voor de IBM 3090.

Als we nu na dit soort hindernissen dan eindelijk de gewenste CPU-tijd, in seconden, gemeten hebben, dan kan de snelheid R_N in Mflops uitgerekend worden met

$$R_N = \frac{k_A * N}{TIJD}\ 10^{-6}\ \text{MFLOPS}$$

waarbij k_A het aantal flops per loop-doorgang aangeeft. Vervolgens wordt voor verschillende waarden van N deze procedure herhaald en via de formule

$$R_N = R(A,N) = R_\infty(A)\ \frac{N}{n_{\frac{1}{2}} + N}$$

levert ons dat evenzoveel lineaire vergelijkingen op voor de parameters $R_\infty(A)$ en $n_{\frac{1}{2}}$. Deze vergelijkingen worden met de kleinste kwadraten methode opgelost. In plaats van $R_\infty(A)$ zullen we in het vervolg gewoon R_∞ schrijven, als duidelijk is welke operatie we bedoelen. In de meeste gevallen zijn de tijdmetingen verricht voor de volgende waarden van N: N = 10(10)100(100)1000(500)10000. Een enkele keer zijn de metingen voor N = 10(10)90 niet meegenomen in de bepaling van R_∞ en $n_{\frac{1}{2}}$ omdat de tijdmeting niet voldoende nauwkeurig kon worden uitgevoerd. In de meeste gevallen werden de loops 500 keer (NREP = 500) uitgevoerd, soms echter 200 keer (NREP = 200).

Met behulp van de gemeten snelheden kan men zich een beeld vormen van de te bereiken rekensnelheden, echter ook niet meer dan een beeld. De werkelijke rekensnelheden voor een algoritme kunnen door verschillende factoren anders uitvallen:

- soms kunnen (tussen)resultaten, die in opeenvolgende loops nodig zijn in de vectorregisters (of in het cachegeheugen) blijven;
- de opstarttijden van gecombineerde loops kunnen elkaar soms overlappen;
- soms leidt de gebruikte datastructuur tot een inefficiënt gebruik van cachegeheugen of treden er niet vermoede memory bank conflicten op.

Men dient dus de hier gepresenteerde gegevens met grote voorzichtigheid te hanteren. De vermelde cijfers zijn redelijk nauwkeurig, zij het dat bij een andere machinebezetting (bijvoorbeeld een lege machine of een overvolle machine) de resultaten wat anders kunnen uitvallen, hetgeen tot kleine wijzigingen in de R_∞ en $n_{\frac{1}{2}}$ waarden kan leiden.

We presenteren nu de R_∞ en $n_{\frac{1}{2}}$ waarden voor een paar bekende vectorcomputers. Voor de volgende eenvoudige DO-loops zijn snelheidsmetingen verricht:

nummer van de loop	aantal operaties per loop-doorgang (k_A)	operatie per loop-doorgang	
1	2	v1(i)=v1(i)+a*v2(i)	('update')
2	2	loop 1 vervangen door SAXPY	(par. 7.2)
3	2	s=s+v1(i)*v2(i)	('inprodukt')
4	2	loop 3 vervangen door SDOT	(par. 7.3)
5	1	v1(i)=v2(i)	(kopiëren)
6	1	v1(i)=v2(ind(i))	('gather')
7	1	v1(i)=v2(ind1(ind(i)))	(dubbele gather)
8	1	v1(ind(i))=v2(i)	('scatter')
9	2	v1(i)=v1(i)+a*v2(ind(i))	(update + gather)
10	1	v1(i)=v2(i)/v3(i)	(deling)
11	1	v1(i)=v2(i)-v1(i-1)	(recursie)
12	2	v1(i)=v2(i)-v3(i)*v1(i-1)	(bidiag. stelsel)
13	1	s=max(s,v1(i))	(max. van vector)
14	9	v1(i)=v2(i)*v0(i)+v3(i)*v0(i-1)+v4(i)*v0(i+1)+ +v5(i)*v0(i-m)+v6(i)*v0(i+m)	(ijle matrix-vector verm.)
15	9	v1(i)=v2(i)*v0(i)+v3(i)*v0(ind(i-1))+v4(i)* v0(ind(i+1))+v5(i)*v0(ind(i-m))+v6(i)* v0(ind(i+m))	(idem als 14, met indir. geadress. vector)
16	12	v1(i)=v2(i)*v3(i)+s*(v2(i)+v3(i))-s2*v4(i) v5(i)=v6(i)+v4(i)+s3*(v6(i)+v4(i))-s4*v3(i)	(meerdere vectoroperaties met opt. mogelijkh.)
17	8	do10j=1,2 10 v1(i)=v1(i)+s1*v2(i,j)+s2*v3(i,j)	(dubbele vector update)
18	8	v1(i)=v1(i)+s1*v2(i)+s2*v3(i)+s3*v4(i)+s4*v5(i)	(viervoudige vector-update)

1. CRAY-1; FORTRAN compiler CFT 77 V1.3 (FORTRAN 77) (februari 1988)

soort loop	loop-nummer	R_∞	$n_{\frac{1}{2}}$
update	1	46	10
update SAXPY	2	45	41
inprodukt	3	75	179
inprodukt SDOT	4	75	121
kopie	5	37	13
gather	6	3.7	15
dubbele gather	7	2	9
scatter	8	4.7	15
update + gather	9	6.3	12
deling	10	13.6	9
recursie	11	10	15
bidiagonaal stelsel	12	10	2
maximum van vector	13	22	141

ijle matrix-vector	14	54	7
ijle matrix-vector met indirect geadresseerde vector	15	5.7	7
meerdere vectoroperaties	16	83	8
dubbele vector-update	17	58	10
viervoudige vector-update	18	77	8

2. CRAY X-MP/2 (8.5 ns versie; 1 processor)

Om de vooruitgang op compilergebied en de afhankelijkheid van de metingen van de compiler te demonstreren, geven we de meetwaarden voor 2 Fortran compilers, namelijk de standaard Fortran compiler CFT 1.15 BF2 en de experimentele versie van de Fortran 77 compiler CFT77 V2.0x207 (februari 1988).

soort loop	loop-nummer	CFT 1.15 BF2		CFT77 V2.0x207	
		R_∞	$n_{\frac{1}{2}}$	R_∞	$n_{\frac{1}{2}}$
update	1	166	84	166	29
update SAXPY	2	190	115	190	106
inprodukt	3	166	292	206	286
inprodukt SDOT	4	206	398	205	303
kopie	5	89	62	105	27
gather	6	54	46	56	17
dubbele gather	7	39	40	36	12
scatter	8	59	45	57	11
update + gather	9	82	44	94	19
deling	10	26	24	31	16
recursie	11	3.5	1	12.4	13
bidiagonaal stelsel	12	5.7	1	15.3	13
maximum van vector	13	47	219	47	184
ijle matrix-vector	14	178	27	174	16
ijle matrix-vector met indirect geadresseerde vector	15	90	16	75	8
meerdere vector operaties	16	170	19	177	10
dubbele update	17	180	56	218	31
viervoudige update	18	202	30	207	13

We zien dat voor verschillende loops de R_∞-waarde is verbeterd en dat voor de meeste loops de $n_{\frac{1}{2}}$-waarde is afgenomen, wat wil zeggen dat de loops eerder op snelheid komen.

3. 2-pipe CYBER 205; Fortran compiler FORTRAN 200 CYCLE 690B (februari 1988)

soort loop	loop-nummer	R_∞	$n_{\frac{1}{2}}$
update	1	200	170
inprodukt	3	100	162
kopie	5	100	97
gather	6	18	25
dubbele gather	7	9.5	28
scatter	8	18	23
update + gather	9	30	46
deling	10	16	20
recursie	11	8	30
bidiagonaal stelsel	12	2.9	1
maximum van vector	13	1.4	1[1]
ijle matrix-vector	14	100	96
ijle matrix-vector met indirect geadresseerde vector	15	23	43
meerdere vectoroperaties	16	120	134
dubbele update	17	200	160
viervoudige update	18	200	160

Opvalllend is het ontbreken van stripmining effecten, in een aantal gevallen wordt de maximale snelheid asymptotisch bereikt (200 Mflops).

4. ETA10-P; Fortran compiler FORTRAN 200 (20 juni 1988)

De werking van de ETA10 processoren is naar buiten toe vrijwel identiek aan die van de 2-pipe CYBER 205. Het meest in het oog lopende verschil is dat een gedeelte van de scalaire kosten van een loop parallel verwerkt kan worden met het echte vectorwerk. Voor sommige eenvoudige loops, zoals de SAXPY (paragraaf 7.2), betekent dit dat de rekentijd voor kleine vectorlengten uitsluitend bepaald wordt door de scalaire kosten, terwijl voor langere vectoren deze kosten gedeeltelijk 'verdwijnen'. Hierdoor is het $(R_\infty, n_{\frac{1}{2}})$-model uit paragraaf 6.2 niet geldig voor kortere vectoren en moet er gerekend worden met formules als $t_A(N) = \max(t_s, (k_A/R_\infty A) * (N + n_{\frac{1}{2}}))$, waarbij t_s de scalaire kosten weergeeft. In de volgende tabel geeft $n_{\frac{1}{2}}$ de vectorlengte weer waarvoor een prestatievermogen $\frac{1}{2} R_\infty$ *gemeten* werd.

[1] Deze loop kan eenvoudig (via VAST) vervangen worden door een loop met $R_\infty = 50$, $n = 161$.

soort loop	loop-nummer	R_∞	$n_{\frac{1}{2}}$
update	1	167	140
inprodukt	3	83.3	243
kopie	5	83.3	108
gather	6	29.5	45
dubbele gather	7	15.2	45
scatter	8	15.2	37
update + gather	9	44	86
deling	10	13.6	21
recursie	11	4.8	27
bidiagonaal stelsel	12	1.6	1
maximum van vector	13	0.73	1
(deze loop kan eenvoudig (via VAST) vervangen worden door een loop met $R_\infty = 41, n_{\frac{1}{2}} = 230$)			
ijle matrix-vector	14	83.3	70
ijle matrix-vector met indirect geadresseerde vector	15	30	48
meerdere vectoroperaties	16	100	68
dubbele update	17	167	83
viervoudige update	18	167	55

Evenals bij de CYBER 205 wordt ook hier, wegens het ontbreken van stripmining, in een aantal gevallen de maximale snelheid (167 Mflops) asymptotisch bereikt. Opvallend is verder de gunstiger verhouding in prestaties voor loops met indirecte adressering ten opzichte van de CYBER 205 (zie bijvoorbeeld loop 7, 9 en 15).

5. IBM 3090/400E met 1 VF; FORTRAN compiler VS level 2.2.0 (maart 1988)

Alle berekeningen zijn uitgevoerd in 'dubbele precisie', dit komt ongeveer overeen met enkele precisie voor CRAY en CYBER. De waarden voor R_∞ en $n_{\frac{1}{2}}$ zijn gemeten voor loops waarbij de vectoren steeds *opnieuw* naar het cachegeheugen gebracht moesten worden. Soms zal men evenwel kans zien vectoren uit het cachegeheugen meerdere malen te gebruiken en dan meet men hogere rekensnelheden (voor vectorlengten die nog precies in het cachegeheugen passen). Deze snelheden zijn aangegeven onder R_{max} (met daarbij n_{max}: de vectorlengte waarvoor ongeveer de snelheid R_{max} gemeten werd).

				alle data in cache	
soort loop	loop-nummer	R_∞	$n_{\frac{1}{2}}$	R_{max}	n_{max}
update	1	19.6	164	28	1500
update SAXPY	2	19.4	133	29	1500
inprodukt	3	19.6	198	27	1500
inprodukt SDOT	4	28	241	44	3000
kopie	5	12	220	21	2000
gather	6	6.5	99	11	1500

soort loop	loop-nummer	R_∞	$n_{\frac{1}{2}}$	R_{max}	n_{max}
dubbele gather	7	5	148	8	1500
scatter	8	3.1	51	15	2000
update + gather	9	10.4	134	15	2000
deling	10	2.7	89	2.8	1500
recursie	11	4	42	7.2	≥10
bidiagonaal stelsel	12	5.4	28	7.4	≥10
maximum van vector	13			7.2	3500
ijle matrix-vector	14	27.7	47	36	500
ijle matrix-vector met indirect geadresseerde vector	15	12.2	23	15	500
meerdere vectoroperaties	16	35.4	47	48	500
dubbele update	17	30	83	42	500
viervoudige update	18	39.1	108	57	500

Over het algemeen vallen de rekensnelheden van de VF tegen, tenzij effectief van het cachegeheugen gebruik gemaakt kan worden voor het bewaren van tussenresultaten, zoals bij loop 16, 17 en 18). Bij indirecte adressering zal het in het algemeen moeilijk zijn om handig met het cachegeheugen om te springen en moet men met lagere snelheden rekening houden.

6. NEC-SX2; Fortran compiler FORTRAN 77/SX, Rev. 036

Ook de NEC-SX2 werkt met een cachegeheugen. Dit tussengeheugen beïnvloedt de rekensnelheid veel minder dan bij de vorige machine. Wanneer de vectoren in het cachegeheugen gehouden kunnen worden, dan ligt de rekensnelheid nauwelijks hoger. We hebben er bij de hier gerapporteerde metingen voor gezorgd dat de betrokken vectoren per loop-doorgang opnieuw van het centrale geheugen naar het cachegeheugen gebracht moesten worden en vice versa.

soort loop	loop-nummer	R_∞	$n_{\frac{1}{2}}$
update	1	548	148
update SAXPY	2		
inprodukt	3	905	607
inprodukt SDOT	4		
kopie	5	369	182
gather	6	63	49
dubbele gather	7	34	36
scatter	8	68	43
update + gather	9	35	30
deling	10	61	50
recursie	11		
bidiagonaal stelsel	12	21	8
maximum van een vector	13	255	692
ijle matrix-vector	14	843	106

soort loop	loop-nummer	R_∞	$n_{\frac{1}{2}}$
ijle matrix-vector met indirect geadresseerde vector	15	102	31
meerdere vectoroperaties	16	740	96
dubbele update	17	683	116
viervoudige update	18	900	108

Opvallende punten in deze prestatievermogenresultaten zijn de lage snelheden die gehaald worden zodra er indirecte adressering in het spel is. De snelheid van de ijle matrix-vector vermenigvuldiging (loop-nummer 14) zakt met een factor 8 zodra de vector indirect geadresseerd is (loop–nummer 15). Merk ook op dat men delingen zoveel mogelijk moet vermijden (evenals trouwens op de andere computers).

7. CONVEX C-1; FORTRAN 77 compiler (26 januari 1988)

Alle berekeningen zijn uitgevoerd in 'dubbele precisie' (overeenkomend met enkele precisie voor CRAY en CYBER).

soort loop	loop-nummer	R_∞	$n_{\frac{1}{2}}$
update	1	6.0	20
update SAXPY	2		
inprodukt	3	6.0	32
inprodukt SDOT	4		
kopie	5	4.5	22
gather	6	2.3	13
dubbele gather	7	1.6	9
scatter	8	2.2	10
update + gather	9	3.1	12
deling	10	1.5	11
recursie	11	0.53	1
bidiagonaal stelsel	12	0.81	1
maximum van vector	13	7.1	41
ijle matrix-vector	14	7.0	17
ijle matrix-vector met indirect geadresseerde vector	15	4.1	8
meerdere vectoroperaties	16	9.4	16
dubbele update	17	7.8	16
viervoudige update	18	11.4	18

Ook bij de CONVEX C-1 is het effect van een tussengeheugen merkbaar, vooral bij operaties met indirecte adressering, bijvoorbeeld loop-nummer 8, 9 en 15. Voor langere vectoren waarvoor tussentijdse verversing van het cachegeheugen nodig is, zakt de snelheid tot respectievelijk 1.5, 2.6 en 3.4 Mflops. Bij de overige loops zagen we dat de rekensnelheid voor vectoren langer dan, zeg, lengte 1000, gelijk werd aan R_∞ en dan verder constant bleef.

8. ALLIANT FX/8; compiler FORTRAN V3.1.33 (2.20 D52) (28 april 1988)
We geven op deze plaats de resultaten van enkele snelheidsmetingen die verricht zijn op één processor (CE), met de in dat geval gunstigst gekozen compiler opties (gv). De snelheid van de ALLIANT met inschakeling van meerdere CE's zal verder in dit boek nog uitvoerig ter sprake komen. Alle berekeningen zijn, net zoals bj de CONVEX C-1, uitgevoerd in dubbele precisie en er is voor gezorgd dat het cachegeheugen voor uitvoering van elke loop ververst werd.

In het voorjaar van 1988 heeft ALLIANT naast haar CE's de zogenaamde ACE's (Advanced CE's) geïntroduceerd. Een machine die met deze ACE's uitgerust is, wordt aangeduid als FX/80. De klokcyclus van de ACE is gelijk aan die van de CE (170 ns), het belangrijkste verschil betreft een gewijzigde floating point chip. We geven hier de meetresultaten voor zowel 1 CE als 1 ACE (voor de ACE is de V4 compiler gebruikt).

		CE		ACE	
soort lus	loop-nummer	R_∞	$n_{\frac{1}{2}}$	R_∞	$n_{\frac{1}{2}}$
update	1	3.1	6	3.5	18
update SAXPY	2				
inprodukt	3	3.9	1	5.1	31
inprodukt SDOT	4				
kopie	5	2.4	9	2.5	25
gather	6	1.5	4	1.9	20
dubbele gather	7	1.0	3	1.3	15
scatter	8	1.5	5	1.9	6
update + gather	9	2.0	4	2.8	15
deling	10	0.5	3	1.1	11
recursie	11	0.85	28	1.6	42
bidiagonaal stelsel	12	0.72	8	1.6	22
maximum van een vector	13	3.1	11	4.7	52
ijle matrix-vector	14	2.0	1	3.6	5
ijle maatrix-vector met indirect geadresseerde vector	15	1.3	1	2.6	4
meerdere vectoroperaties	16	2.7	1	6.1	6
dubbele update	17	4.2	5	4.6	13
viervoudige update	18	5.1	2	5.6	6

We zien dat de snelheden ruwweg een factor 2 lager liggen dan bij de CONVEX C-1, hetgeen op grond van de klokcyclus ook verwacht mocht worden. We merken verder op dat de ACE in vele gevallen een belangrijke snelheidsverbetering geeft ten opzichte van de CE.

Uitvoerige metingen voor complete algoritmen en voor vrij realistische programma's kan men vinden in een serie rapporten uitgegeven door het Academisch Computer Centrum Utrecht [39,40,41,42,58]. Ook in het tweemaandelijks uitgegeven tijdschrift *Supercomputer* (red. A. Emmen, J. Hollenberg, R. Llurba en A. van der Steen) kan men veel van dit soort gegevens vinden.

ZEVEN

EENVOUDIGE VECTORISEERBARE ALGORITMEN

In dit hoofdstuk gaan we de verwerking van een paar eenvoudige, voor vectorverwerking geschikte, rekenschema's op verschillende vectorcomputers proberen te begrijpen en zullen we ons een indruk proberen te verschaffen van de mogelijkheden die verwerking met vectoroperaties te bieden heeft. We zullen daarbij zien dat met de eerder gegeven karakteristieken (zoals klokcycluslengte, load- en store-mogelijkheden, registers en chaining) de werkelijk te behalen snelheden van de machines voor een gegeven algoritme redelijk benaderd kunnen worden. Meerdere malen blijkt dat de gemeten snelheid beneden de verwachting blijft, een enkele maal komt het ook voor dat de gemeten snelheid de aanvankelijke verwachting overtreft.

Hier raken we al meteen een belangrijk punt. We moeten ons steeds terdege blijven realiseren dat we met onze analyses de mogelijkheden van de hardware verkennen, terwijl de gemeten snelheden in feite slechts weergeven wat de betreffende FORTRAN compiler aan mogelijkheden in een programma onderkent en benut. De invloed van de compiler wordt aardig geïllustreerd door de snelheidsmetingen voor de CRAY X-MP in paragraaf 6.3. De metingen, en de conclusies daaruit omtrent opstarttijden en strip-mining-overhead, geven dan ook veel meer een momentopname te zien dan de werkelijke mogelijkheden van een machine.

Regelmatig zien we het gebeuren dat met een nieuwe compiler versie de verwerkingssnelheid voor sommige constructies toeneemt en dat ligt dan meestal niet aan veranderingen aan de hardware. Evenwel komt het ook voor dat door aangebrachte wijzigingen in de compiler of in de hardware de zorgvuldige timing van vectoropdrachten in sommige gevallen teniet gedaan wordt waardoor (tijdelijk, tot de volgende compiler versie) een verslechtering van de snelheid kan optreden.

We zullen in dit hoofdstuk steeds uitgaan van de mogelijkheden die de hardware biedt en dan, waar nodig en/of interessant, de feitelijk waargenomen situatie van commentaar voorzien.

Bij de analyse van algoritmen richten we de aandacht eerst op de binnenste DO-loops. De daarin optredende sets geïndiceerde variabelen zullen we aanduiden als vectoren. Met name operaties op kolommen of rijen van een matrix zullen door ons als vector-operaties gekenschetst worden.

Het is wellicht goed er nu reeds de aandacht op te vestigen dat de snelheden voor diverse vector-algoritmen heel verschillend kunnen zijn, zelfs voor één en dezelfde computer. De verschillende computers die we in onze analyses betrekken zijn vrij willekeurig gekozen, we hebben ons echter trachten te beperken tot architecturen die ten opzichte van elkaar wat nieuwe gezichtspunten opleveren en voorts hebben we ons beperkt tot die machines waarop we zelf tijdmetingen hebben uitgevoerd.

7.1 Het bijwerken van een vector (de vector-update)

Een groot aantal gecompliceerde algoritmen uit de lineaire algebra kan beschreven worden in termen van slechts een beperkte set eenvoudige vectoroperaties. Deze vectoroperaties zijn in FORTRAN gedefiniëerd als de zogenaamde BLAS [43], hetgeen staat voor Basic Linear Algebra Subprograms. Een belangrijk lid van deze set is de SAXPY operatie, dat is een operatie van de vorm $y_i = a * x_i + y_i$, $i = 1,....,N$, a een constante, (ofwel A maal X Plus Y, de S staat voor Single precision). Deze SAXPY, of vector-update, speelt een belangrijke rol in bijvoorbeeld het oplossen van stelsels lineaire vergelijkingen via directe methoden (gauss-eliminatie) danwel iteratieve methoden.

Soms komt men ook de variant $z_i = a * x_i + y_i$ tegen, dat wil zeggen dat de resultaatvector z verschilt van y. Deze constructie behoort (helaas) niet tot de BLAS-set, maar in onze analyses kunnen we haar gelukkig over één kam scheren met de echte SAXPY.

De SAXPY valt uiteen in de volgende elementaire vectoroperaties:

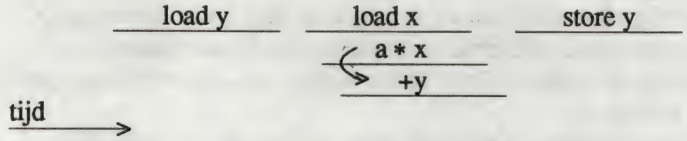
- 2 vector loads (x en y);
- 1 vector store (y, of indien gewenst: z);
- 1 vectorvermenigvuldiging (met de constante a, waarvan we aannemen dat die vooraf in een geschikt register geladen wordt; de tijd daarvoor wordt in de opstarttijd voor SAXPY verdisconteerd);
- 1 vectoroptelling (namelijk van y met het resultaat van a * x)

Per waarde van i worden er dus twee flops uitgevoerd: $k_A = 2$.

We bekijken nu voor een paar representatieve machines hoe het een en ander uitpakt.

7.1 a. CRAY-1

De CRAY-1 heeft slechts 1 toegang tot het geheugen per cyclus van 12.5 ns, maar laat verder wel chaining van operaties toe, waardoor afgezien van stripmining-effecten, de SAXPY in de tijd gezien als volgt kan worden uitgevoerd:



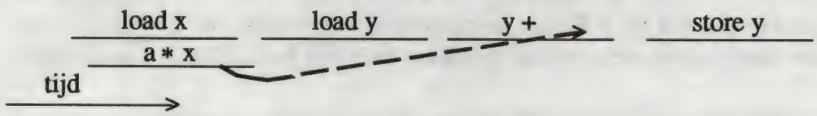
(Voor vectoren langer dan de vectorregisterlengte moet het hierboven geschetste schema cyclisch herhaald worden, we zullen dat niet steeds expliciet vermelden.)

Afgezien van opstarttijden beslaat ieder tijdsegment N klokcycli en dus $k_c = 3$. De asymptotische snelheid voor een SAXPY zou daarom gelijk kunnen zijn aan

$$R\,(SAXPY) = (k_A/k_c) * 80 \approx 53 \text{ Mflops.}$$

In werkelijkheid hebben we echter (voor zeer lange vectoren) een topsnelheid van ongeveer 44 Mflops waargenomen, hetgeen betekent dat k_c vervangen moet worden door $\tilde{k}_c \approx 3.65$ (zie paragraaf 6.2). Deze verhoging van k_c met 0.65 is een gevolg van stripmining en we zien dat die dus een overhead van ongeveer 20% met zich meebrengt. Deze stripmining heeft kennelijk tot gevolg dat de 3 tijdsegmenten elkaar niet vloeiend kunnen opvolgen bij het verwerken van opeenvolgende parten van de vectoren.
De $n_{\frac{1}{2}}$-waarde (zie paragraaf 6.3) ligt in de buurt van 10 en dat ligt belangrijk lager dan wat Hockney waarnam (voor een andere compiler-versie!) [32].

De invloed van de kwaliteit van een compiler op de rekensnelheid laat zich aardig illustreren met de volgende historie. Bij de eerste CRAY FORTRAN compilers verliep de verwerking van de SAXPY, in de tijd gezien, als volgt:



dus $k_A = 2$, $k_c = 4$, en $R_\infty \approx 40$ Mflops.

Doordat het segment waarin a * x wordt uitgerekend het tijdsegment waarin y geladen wordt lichtjes overlapt, ging de mogelijkheid om het optellen bij y te koppelen met het laden van y verloren. Door nu in FORTRAN de SAXPY te programmeren met extra haakjes rond y_i, zoals bijvoorbeeld in

```
      DO 10 I = 1,N
10    Y(I) = (Y(I)) + A * X(I)
```

werd de compiler geforceerd eerst Y te laden, waardoor dan de in de aanhef beschreven verwerking ontstond. Dit is de reden waarom men in oudere publicaties over metingen op de CRAY-1 af en toe deze schijnbaar overtollige haakjes ziet optreden.

7.1 b. CONVEX C-1 en ALLIANT FX/8

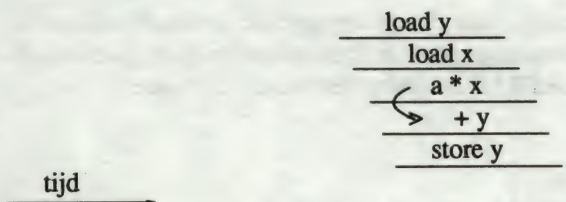
De CONVEX C-1 verschilt, voor onze analyses althans, van de CRAY-1 voornamelijk door de lengte van de klokcyclus (100 ns voor een CONVEX tegen 12.5 ns voor CRAY). Dat zou met $k_A = 2$ en $k_c = 3$ kunnen leiden tot een asymptotische snelheid voor SAXPY van $R_\infty \approx 6.7$ Mflops. In werkelijkheid meten we, ten gevolge van stripmining, een maximale snelheid die ongeveer 6 Mflops bedraagt. Dat betekent dat er voor grote vectorlengtes N in plaats van 3 ongeveer 3.4 klokcycli per 2 flops worden verbruikt ($\tilde{k}_c \approx 3.4$), hetgeen een stripmining overhead van ongeveer 18% inhoudt (nagenoeg hetzelfde percentage als voor de CRAY-1). De waarde van n waarvoor de helft van de maximumsnelheid wordt behaald, $n_{\frac{1}{2}}$, bedraagt ongeveer 20 en dat is iets ongunstiger dan voor de CRAY-1.

Voor één CE van een ALLIANT FX/8 zou men op grond van de klokcycluslengte (170 ns) een asymptotische snelheid mogen verwachten van 2/3 * 5.88 = 3.92 Mflops. Gemeten werd evenwel $R_\infty \approx 3.1$ Mflops, met $n_{\frac{1}{2}} \approx 6$. Het verschil tussen gemeten topsnelheid en de theoretische top moet volgens [42] gezocht worden in stripmining. Vermoedelijk speelt ook het tussengeheugen (cache) hierbij een rol. Voor de nieuwe ACE-processor werd een iets betere prestatie gemeten: $R_\infty \approx 3.5$ Mflops, met $n_{\frac{1}{2}} \approx 18$.

7.1 c. CRAY X-MP

We beschouwen hier het gebruik van slechts 1 processor van de X-MP. Op het parallelle gebruik van meerdere processoren komen we later nog terug.

Gezien de mogelijkheid om simultaan 2 vectorelementen te laden en er eentje weg te schrijven per klokcyclus, alsmede chaining, zou de verwerking van de SAXPY er schematisch als volgt uit kunnen zien:



Afgezien van stripmining zouden we gezien de klokcyclus van 9.5 ns en $k_A = 2$, $k_c = 1$ rekening mogen houden met een maximumsnelheid ter grootte van $R_\infty = 2 * 105 = 210$ Mflops (en van 235 Mflops bij modellen met een klokcyclus van 8.5 ns). Hockney [32] rapporteert evenwel een *gemeten* waarde R_∞ (SAXPY) ≈ 148 Mflops voor de 9.5 ns processor en dat stemt overeen met eigen oude metingen. Dat hier heel duidelijk de kwaliteit van de FORTRAN compiler een rol speelt blijkt wel uit de belangrijk hogere waarden die tegenwoordig gehaald worden (zie paragraaf 6.3) en uit het feit dat de vanuit FORTRAN aan te roepen *subroutine* SAXPY, die precies doet wat wij willen, het karwei klaart met een *gemeten* maximumsnelheid R_∞ (SAXPY) ≈ 190 Mflops (deze metingen zijn verricht op modellen met een 8.5 ns klokcyclus). Blijkens persoonlijke mededelingen van CRAY-medewerkers moet het resterende snelheidsverschil van 190 ten op-

zichte van 235 Mflops geheel aan op dit moment nog onvermijdelijke stripmining worden toegeschreven.

De invloed van een zorgvuldige timing van de operaties (chaining luistert namelijk nogal nauw) blijkt hieruit dat na het beschikbaar komen van de modellen met een klokcyclus van 8.5 ns de eerste FORTRAN compiler (dezelfde als voor het 9.5 ns model) het niet verder wist te schoppen dan een povere 100 Mflops. Dit euvel werd vrij spoedig verholpen.

7.1 d. CYBER 205

In Control Data CYBER-terminologie is de SAXPY-operatie een representant van hun zogenaamde 'linked-triad' constructie (zie onze eerdere beschrijving van de CYBER 205) en kan dus, na een zekere opstarttijd, worden uitgevoerd met de snelheid van 1 resultaat per klokcyclus. Met $k_c = 1$ en met $k_A = 2$ leidt dat tot R_∞ (SAXPY) = $100n_p$, waarbij we met n_p het aantal vectorpipes aangeven. De CYBER 205 heeft geen vectorregisters en zolang de vectorlengten maar kleiner blijven dan N = 65535, spelen stripmining effecten geen enkele rol. Voor langere vectoren worden dan ook inderdaad snelheden gemeten die dicht bij R_∞ liggen (zie paragraaf 6.3).

Zoals we ook al eerder meldden ligt de opstarttijd bij de CYBER 205 relatief nogal hoog en dat leidt dan automatisch ook tot hoge waarden voor $n_{\frac{1}{2}}$. Voor de 2-pipe machine hebben we voor SAXPY een waarde van om en nabij de 170 gemeten (zie paragraaf 6.3).

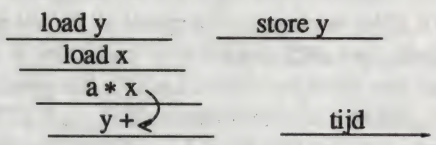
De ETA10-processoren laten zich op vrijwel identieke wijze als de CYBER 205 analyseren. De uit metingen verkregen waarden van R_∞ voor de SAXPY zijn dan ook gelijk aan de betreffende maximumsnelheden van de machines (zie paragraaf 6.3). Door meer overlappingsmogelijkheden van vectorieel en scalair werk worden wel vaak lagere $n_{\frac{1}{2}}$-waarden bereikt.

We merken tot slot nog op dat de linked triad een speciale hardware constructie van de CYBER 205 en de ETA10 is en de gemeten snelheden worden dus op *geen enkele wijze* door de kwaliteit van de FORTRAN compiler (ten nadele) beïnvloed.

7.1 e. FUJITSU FACOM VP-200

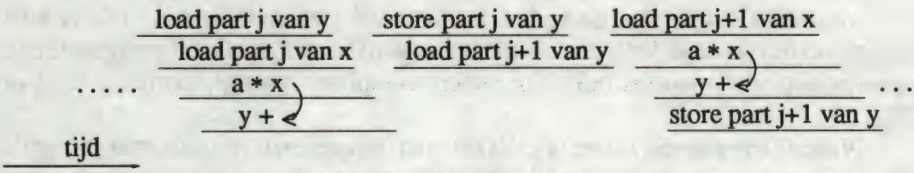
De tot nu toe behandelde machines hadden allen wel een of andere eigenaardigheid die in het licht van een snelheidsanalyse het vermelden waard was, uitgezonderd de CONVEX-machine wellicht die qua gedrag wel heel dicht bij de CRAY-1 staat. De VP-200 vormt hierop geen uitzondering.

Door de 2 load/store mogelijkheden (voor vectorelementen) per klokcyclus kan de verwerking van de SAXPY operatie er als volgt uitzien:



Dus, onder verwaarlozing van stripmining effecten hebben we $k_A = 2, k_c = 2$ en dat leidt tot een theoretische top van $R_\infty \approx 267$ Mflops.

In werkelijkheid hebben we echter snelheden gemeten van om en nabij de 300 Mflops. Verbazing was ons deel! De FUJITSU FORTRAN compiler, het mag nog wel eens gezegd worden, is van hoog niveau. Voorbeeld: als de vectorlengte tijdens compilatie expliciet bekend is (en dat was zo in onze testsituatie) en als deze bovendien groot is, dan worden de vectoren in parten ter lengte 1024 gezaagd. De verwerking van deze parten komt er schematisch als volgt uit te zien:



De stripmining wordt hier heel slim uitgevoerd. Gemiddeld worden er, afgezien van allerlei overhead, 4 flops per 3 klokcycli uitgevoerd: $k_A = 2, k_c = 1.5$; het slecht benutte tijdsegment uit het eerste diagram wordt nu dus over twee parten verdeeld. Dit leidt tot een theoretische topsnelheid van $R_\infty = 4/3 * 267 = 356$ Mflops.

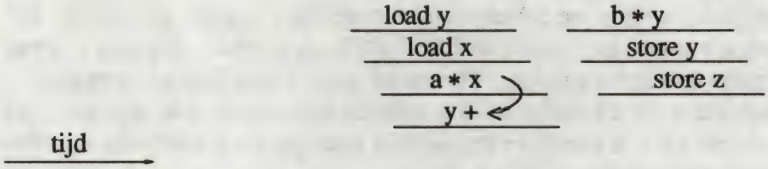
Ons model is nu niet meer toereikend om het verschil met de werkelijk gemeten topsnelheid van ongeveer 300 Mflops te verklaren. Afgezien van de stripmining zelf, zal het duidelijk zijn dat een vectorlengte die een geheel veelvoud van 1024 bedraagt ideaal is omdat er dan geen partje met een 'half benut' segment overblijft.

Als de waarde van N tijdens compilatie niet bekend is, meten we snelheden tot ongeveer 250 Mflops en kennelijk wordt dan de 'overlap'-mogelijkheid onbenut gelaten. Echter ook in deze situatie is het weer moeilijk om eenvoudige uitspraken over (negatieve) stripmining effecten te doen. Als men bijvoorbeeld weet dat (in een deel van het algoritme) de vectorlengten beperkt blijven (zeg kleiner dan 1024) dan kan de programmeur dat in het programma kenbaar maken. De vectorregisters worden dan geherconfigureerd tot een paar registers van de gewenste lengte, waardoor stripmining geheel wordt uitgeschakeld. Dat dit eventueel tot een behoorlijke besparing zou kunnen leiden werden we reeds gewaar bij de discussie van de SAXPY op de CRAY-1 en de CONVEX C-1, waar de overhead in de buurt van de 20% lag. Metingen op de VP-200 suggereren ook inderdaad soortgelijke besparingen in de rekentijd. We moeten hier wat vaag blijven omdat het effect opgemerkt werd in een situatie waarin SAXPY onderdeel uitmaakte van een veel omvangrijker algoritme en niet als zodanig apart getimed werd.

De analyse van eenvoudige algoritmen (zoals SAXPY) wordt op de VP-200 nog verder bemoeilijkt doordat de compiler, indien mogelijk, vectoropdrachten samenneemt om ook aldus de verwerkingstijd verder te reduceren. Als we bijvoorbeeld de SAXPY in combinatie aantreffen, zoals in

```
      DO 10 I = 1, N
      Y(I) = Y(I) + A * X(I)
10    Z(I) = B * Y(I)
```

dan wordt het verwerkingsschema:

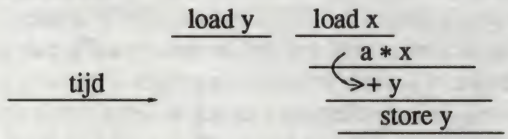


We constateren dat, wellicht afgezien van wat extra overhead, de tijd die nodig is voor deze constructie, gelijk is aan die voor de SAXPY zelf. Het uitvoeren van $z_i = b * y_i$ kost dus geen extra rekentijd, hetgeen door metingen werd bevestigd. (Overigens gebeurt dit samennemen van operaties ook door andere compilers zoals bijvoorbeeld die van CRAY.)

Vermeldenswaardig is hier nog dat het juist interpreteren van tijdmetingen, op de VP-200 verricht, geen sinecure is. De FORTRAN compiler voert namelijk allerlei optimalisaties uit waar de gebruiker niet altijd op bedacht is. Geeft men, om maar een voorbeeld te noemen, in een testprogramma aan de elementen van een vector allemaal dezelfde waarde, dan 'ziet' de compiler dat en maakt van die wetenschap later gebruik. Voert men een bepaald algoritme 1000 keer uit om een (statistisch) nauwkeuriger tijdmeting te verkrijgen, dan heeft de compiler dat ook al ras in de gaten met alle (voor een tijdmetingsprogramma) ongewenste gevolgen vandien.

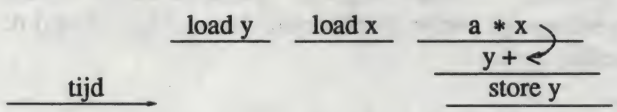
7.1 f. NEC SX-2

De NEC SX-2 is in staat om per vector pipe twee vectorelementen (van *dezelfde* vector) per klokcyclus te laden en er 1 op te bergen. In principe zou dat, ook volgens [28] moeten leiden tot



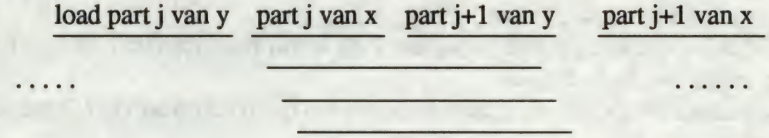
Deze wijze van verwerking zou, wegens $k_A = 2$, $k_c = 1.5$, uitzicht bieden op R_∞ (SAXPY) = 2/1.5 * 667 ≈ 889 Mflops.

In de praktijk werd echter een waarde van R_∞ = 576 Mflops gemeten (zie paragraaf 6.3 en [61]). Dit leidt tot de verdenking dat er de volgende schematische verwerking plaatsvindt:



Deze wijze van verwerking zou leiden tot R_∞ = 667 Mflops, hetgeen beter strookt met de gemeten waarde van 569 Mflops. Het resterende verschil zou dan aan stripmining kunnen worden toegeschreven.

Er is nog een derde mogelijkheid. Er kan, althans in principe, hier ook partiële overlapping van het (*uit*stekende) 'load y' segmentje plaatsvinden, waardoor na verdeling van de vectoren in parten van geschikte lengte, de verwerkingscyclus er als volgt zou kunnen uitzien:



Dit schema leidt tot een R_∞ in de buurt van 1333 Mflops.

We hebben nu drie verschillende verwerkingsmogelijkheden gezien, waarvan de tweede, op grond van de waargenomen snelheid, de meest waarschijnlijke lijkt. Definitief uitsluitsel wordt verschaft door de rekensnelheden te analyseren met het $(R_\infty, n_{\frac{1}{2}})$-model inclusief stripmining effecten (zie paragraaf 6.2). Het blijkt dan dat $k_c = 1$ (dus de *derde* mogelijkheid wordt in werkelijkheid gebruikt) en de lage praktijkwaarde voor R_∞ moet geheel aan stripmining effecten worden toegeschreven.

Ook hier ligt de zaak in werkelijkheid weer gecompliceerder (net als bij VP-200, omdat ook hier bij beperkte (vooraf bekende) vectorlengten de stripmining kan worden uitgeschakeld.

7.1 g. IBM 3090 met VF

In principe heeft de VF van IBM voor een SAXPY ook ongeveer drie klokcycli van 18.5 ns nodig per resultaat, immers ook hier is slechts één geheugencontact (hetzij lees, hetzij schrijf) *met cachegeheugen* per klokcyclus mogelijk. Wanneer de betrokken vectoren in het cachegeheugen zijn en het resultaat daar kan blijven, dan zou men voor redelijk grote vectorlengte een snelheid van ongeveer 1000/18.5 * 2/3 = 36 Mflops mogen verwachten.

Zoals blijkt uit de metingen in paragraaf 6.3 is voor vectoren die zich permanent in het cachegeheugen bevinden ongeveer 29 Mflops gemeten. Stripmining en een vrij hoge opstartwaarde (grote $n_{\frac{1}{2}}$) kunnen het verschil met de asymptotische waarde van 36 Mflops verklaren. Voor vectoren die eerst nog uit het centrale geheugen gehaald moeten worden, zakt de snelheid verder tot 19.4 Mflops (voor grote N). Klaarblijkelijk sluit de geheugensnelheid niet voldoende aan op de functional unit snelheid.

7.2 Het inprodukt op vectorsnelheid

Bij wetenschappelijk rekenwerk komt het vaak voor dat er een inprodukt berekend moet worden. Geen wonder dat alle fabrikanten van supercomputers er voor hebben gezorgd dat dit op snelle wijze kan gebeuren. We gaan wat dieper in op de manieren waarop ze

dat hebben trachten te realiseren omdat dit ons wat inzicht verschaft in de wijze waarop algoritmen gevectoriseerd kunnen worden.

Het inprodukt tussen twee vectoren x en y wordt gedefiniëerd als $(x,y) = \sum_{i=1}^{N} x_i y_i$, dat wil zeggen dat het gelijk is aan de som van de elementsgewijze produkten. Uiteraard kan het inprodukt ook berekend worden tussen twee willekeurige segmenten $(x_p,....,x_q)$ en $(y_k,....,y_m)$, mits m - k = q - p, als $\sum_{i=1}^{q-p+1} x_{p+i-1}\, y_{m+i-1}$

In BLAS-terminologie (zie paragraaf 7.4) wordt het inprodukt aangeduid als SDOT en wij zullen dat ook doen.

Een stukje FORTRAN-programma waarin een inprodukt wordt berekend zou er als volgt uit kunnen zien:

```
      SDOT = 0.
      DO 10 I = 1, N
10    SDOT = SDOT + X(I) * Y(I)
```

Algoritme 7.2.1 Standaardversie van SDOT

Op grond van wat we inmiddels weten van vectorverwerking lijkt deze code niet bar geschikt voor gesegmenteerde functional units, immers voor het bijwerken van SDOT in de j-de stap hebben we de waarde van de (j-1)-ste stap nodig en daar wordt naar alle waarschijnlijkheid nog ergens in de optel functional unit aan gesleuteld. De vroegste CRAY FORTRAN compilers verwerkten deze code dan ook op scalaire snelheid; tegenwoordig 'herkennen' alle moderne compilers het stukje code als een inprodukt-berekening en wordt het door de compilers keurig vervangen door een efficiënte object-code die hoort bij een goed vectoriseerbaar algoritme.

Ook hier is weer het punt aangebroken om op een principieel punt te wijzen. Bij de berekening via algoritme 7.2.1 hoort een vast afrondfoutenpatroon, het eindantwoord bevat een fout ten gevolge van het rekenen in eindige precisie. Als je nu in vol vertrouwen dit algoritme voor uitvoering opdraagt aan de computer, dan is het misschien helemaal niet zo leuk als diezelfde computer dat niet serieus neemt en 'in het geniep' het inprodukt op een hele andere wijze berekent (mogelijk met een geheel andere fout-opbouw). Nu loopt het met het het inprodukt niet zo'n vaart, de alternatieve schema's leveren een vergelijkbaar goed antwoord op, voor andere algoritmen zou dat best heel anders kunnen uitpakken. Men zij dus gewaarschuwd!

Hoe ziet nu zo'n wel-vectoriseerbaar algoritme er uit? Er zijn een paar mogelijkheden, echter de voor vectorregistermachines meest gebruikte variant is de volgende, waarvan we het recept geven:

---

a. reserveer een (leeg) vectorregister, we noemen dit z (met leeg bedoelen we dat er nullen in staan);
b. hak de vectoren x en y in parten ter lengte van een vectorregister. We duiden deze parten aan met $\overline{x}_j$ en $\overline{y}_j$ en veronderstellen dat er m parten zijn (het laatste partje van elk is wellicht wat korter!);

c. voor deze m parten doen we het volgende:
laad $\bar{x}_j$ en laad $\bar{y}_j$,
bereken de produkten tussen de elementen van $\bar{x}_j$ en $\bar{y}_j$
en tel deze op bij z.

d. er staan nu een aantal partiële sommen in z en alles wat we nog moeten doen is de elementen van z bij elkaar optellen. Dat kan ook weer slim door bijvoorbeeld eerst de tweede helft van z bij de eerste helft op te tellen en zo verder; we laten dit verder onbesproken.

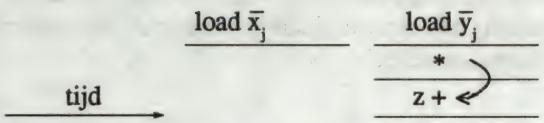
Algoritme 7.2.2 Vectorregisterversie voor SDOT

Men ziet snel in dat wanneer N niet al te klein is, al het werk gaat zitten in stap c en deze stap is volledig vectoriseerbaar. Stap a hoeft vaak niet uitgevoerd worden, we beginnen in stap c gewoon met z elementsgewijs gelijk te maken aan de produkten van $\bar{x}_1$ en $\bar{y}_1$ en stap b wordt in feite voor ons gedaan via de zogenaamde stripmining. Stap d tenslotte beschouwen we maar als overhead.

We bespreken nu weer de berekening van SDOT op een paar uiteenlopende machines, waarbij we vooral aandacht schenken aan de in het oog lopende afwijkingen.

7.2 a. CRAY-1 en CONVEX C-1

Stap c van algoritme 7.2.2 wordt op beide machines schematisch als volgt uitgevoerd:



Voor ons rekenmodel voor de snelheid volgt $k_A = 2$ en $k_c = 2$. Afgezien van opstarttijden, stap d en stripmining, mogen we dus op de CRAY-1 een limietsnelheid verwachten ter grootte van R_∞ (SDOT) = 80 Mflops.

Uit de werkelijk gemeten snelheden R(SDOT, 1000) = 63 Mflops en R (SDOT, 5000) = 72 Mflops (zie [58]) kunnen we de modelwaarden $n_{\frac{1}{2}}$ en R_∞ (SDOT) bepalen:

$$n_{\frac{1}{2}} \approx 185 \text{ en } R_\infty(\text{SDOT}) \approx 75 \text{ Mflops},$$

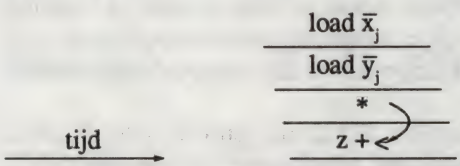
en dit komt prachtig overeen met de gemeten waarden in paragraaf 6.3.

De vrij hoge waarde voor $n_{\frac{1}{2}}$ lag al wel min of meer voor de hand, immers voor lage waarden van N drukt het sommeren van de partiële sommen in z (stap d) natuurlijk vrij zwaar op de snelheid. Stripmining blijkt slechts een geringe invloed op het prestatievermogen te hebben gezien het geringe verschil tussen de theoretische limietsnelheid en de uit metingen afgeleide limietsnelheid. Tegenwoordig ligt, met de nieuwste compilers, de snelheid voor het inprodukt op de CRAY-1 nog weer iets gunstiger (zie paragraaf 6.3: een lagere $n_{\frac{1}{2}}$-waarde).

Voor de CONVEX C-1 bepalen we uit de meetwaarden (zie [42]) R(SDOT, 500) = 5.95 Mflops en R(SDOT, 1000) = 6.12 Mflops, dat $n_{\frac{1}{2}} \approx 30$ en R_∞ (SDOT) = 6.3 Mflops en dat klopt ook vrij aardig met de metingen in paragraaf 6.3. Voor lage waarden van N voldoet ons model niet zo goed, omdat we [N/r] vervangen hebben door N/r, en de werkelijke waarde voor $n_{\frac{1}{2}}$ ligt dan ook ergens bij 45. Desalniettemin blijkt dat de C-1 al vroeg op snelheid komt. De 'gemeten' limietsnelheid steekt evenwel pover af bij de waarde die we op grond van $k_A = 2$, $k_c = 2$ zouden mogen verwachten, namelijk 10 Mflops. De oorzaak van het verschil is ons nog niet duidelijk (invloed van het cache-geheugen?).

7.2. b. CRAY X-MP

Doordat er hier twee simultane loadunits voorhanden zijn, ligt de verwerking van stap c in algoritme 7.2.2 wel min of meer voor de hand:



De waarden $k_A = 2$, $k_c = 1$ leiden hier tot een theoretisch te behalen piekwaarde R_∞ (SDOT) = 210 Mflops (op 1 processor met een 9.5 ns klokcyclus). Op grond van de werkelijk gemeten snelheden (zie [39]) voor N = 1000 (111.1 Mflops) en N = 5000 (169.5 Mflops) kunnen we de waarden $n_{\frac{1}{2}} \approx 750$ en R_∞ (SDOT) = 195 Mflops bepalen. De eerste waarde duidt op een wel zeer hoge overhead (ten gevolge van stap d?), terwijl de tweede waarde laat zien dat stripmining hier een marginale invloed heeft. Voor meer recente en gunstiger meetwaarden zie paragraaf 6.3.

7.2 c. FUJITSU VP-200

Voor de VP-200 zou men natuurlijk een zelfde schematische verwerking mogen verwachten als voor de X-MP en dat zou dan leiden tot een theoretische grens van 533 Mflops. De werkelijk gemeten tijden blijven hier belangrijk onder en dat vindt zijn oorzaak in een afwijkende berekening van het inprodukt [53]:

---

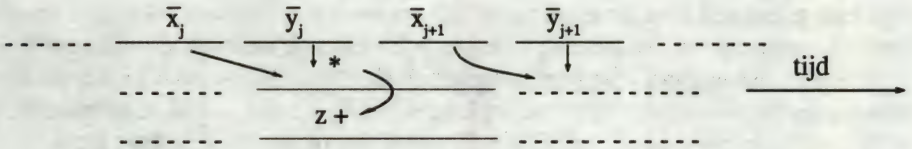
a. de vectoren x en y worden geladen en elementsgewijs met elkaar vermenigvuldigd;
b. de elementsgewijze produkten worden vervolgens gesommeerd met de hardware instructie VSUM (die de som van de elementen van een vector bepaalt). Deze VSUM wordt dan wel gekoppeld met stap a, de verwerking loopt toch uit ten opzichte van stap a tot ~ 1.5 N klokcycli.

Algoritme 7.2.3 SDOT berekening op de VP-200

Een waarde van $k_A = 3/2$, met $k_c = 2$, suggereert dan een limietsnelheid van 400 Mflops en dat komt redelijk overeen met de uit de in [39] gerapporteerde waarden (R(SDOT, 1000) = 333 Mflops, R(SDOT, 5000) = 380 Mflops) bepaalde R_∞ (SDOT) ≈ 380 Mflops en $n_{\frac{1}{2}} \approx 145$. Kennelijk leidt de afwijkende inproduktberekening tot minder opstart overhead en misschien geeft dat dan een (gering) voordeel voor kortere vectorlengten ten opzichte van algoritme 7.2.2. We hebben dit niet verder onderzocht.

7.2 d. NEC SX-2

Zoals reeds bij de SAXPY werd aangegeven, kan de afwijkende load mogelijkheid bij de SX-2 benut worden om porties van de vectoren alvast in 'het voren' te laden:



Met $k_A = 2$, en $k_c = 1$ (gemiddeld) mogen we een topsnelheid in de buurt van 1333 Mflops verwachten. Daadwerkelijke metingen leiden tot een realistische top van R_∞ (SDOT)= 952 Mflops (zie paragraaf 6.3).

7.2 e. CYBER 205

Op de CYBER 205 wordt het inprodukt berekend via de Q8SDOT instructie. Dat is een in microcode vastgelegd berekeningsschema voor het berekenen van de elementsgewijze produkten van x en y. Deze produkten vloeien echter niet terug naar het geheugen maar worden geaccumuleerd in een paar scalaire registers. De inhoud daarvan wordt tenslotte opgeteld en dat levert de gewenste inprodukt-waarde. Het lijkt wat op de aanpak bij de VP-200 (algoritme 7.2.3), echter hier vindt het accumuleren kennelijk plaats op volle snelheid (bij de VP-200 'verloren' we 1/2 N klokcycli bij het optellen) want *metingen* leiden tot R_∞ (SDOT) = 100 Mflops (zie paragraaf 6.3). Deze waarde zouden we inderdaad als bovengrens verwachten op grond van $k_A = 2$ en $k_c = 1$ (met $n_p = 1$). Omdat de vector pipes niet gebruikt worden, is de snelheid van het inprodukt onafhankelijk van het werkelijke aantal vector pipes waarmee een machine is uitgerust.

Een alternatief berekeningsschema is gesuggereerd door Schreiber & Tang [52]. Als we voor het gemak even aannemen dat N een macht van 2 is, dan loopt hun algoritme zoals op de volgende pagina is weergegeven.

a. Bereken $z_i = x_i * y_i$, i = 1,....,N;
b. Tel de tweede helft van z op bij de eerste helft:
   $z_i = z_i + z_{N/2+i}$, i = 1,....,N/2
   Herhaal deze stap voor de resterende 'halve' vector:
   $z_i = z_i + z_{N/4+i}$, i = 1,....,N/4
   En zo voort totdat het 'vectortje te klein is geworden'.
   Dat resterende vectortje representeert dan een paar partiële sommen en die tellen we scalair bij elkaar op.

Algoritme 7.2.4 'Halverings' algoritme voor SDOT

Men gaat gemakkelijk na dat stap a op de 205 ongeveer N (effectieve) klokcycli vergt. De eerste optelstap van fase b neemt ongeveer N/2 klokcycli, de volgende ongeveer N/4 en zo verder, dus in totaal kost stap b ongeveer N/2 + N/4 + N/8 + ...≈N klokcycli. Op deze manier hebben we een algoritme verkregen met $k_A = 2$ en $k_c = 2$ en dus een theoretische snelheidsgrens van R_∞ (SDOT) = 50 n_p Mflops (n_p = aantal vector pipes).

We zien onmiddellijk dat we op een 1-pipe's machine (bestaan die eigenlijk wel? Die van SARA bleek achteraf een beperkt geïnstalleerde 2-pipe's te zijn) beter af zijn met de Q8SDOT-instructie. Op een 2-pipe's machine speelt algoritme 7.2.4, althans theoretisch, quitte, echter op een 4-pipe's machine (waarvan er vermoedelijk slechts twee bestaan) levert algoritme 7.2.4 winst op door twee keer zo snel te zijn (althans voor grote N).

Tenslotte zij nog vermeld dat Q8SDOT ook een lange aanlooptijd nodig heeft, eerst bij $n_{\frac{1}{2}} \approx 160$ zitten we op de halve snelheid van 50 Mflops.

7.3 De volle matrix-vector vermenigvuldiging

De in de voorgaande paragrafen uitvoerig besproken SAXPY en SDOT operaties staan in een programma zelden op zich maar maken veelal deel uit van complexere algoritmen. Bij het analyseren van zo'n algoritme kan men natuurlijk deze onderdelen met een gerust geweten overslaan, omdat we inmiddels gezien hebben dat ze vrij probleemloos vectoriseerbaar zijn en men kan dan de aandacht verder bepalen op de overige bouwstenen van het algoritme. Echter, zoals zal blijken uit de navolgende bespreking van de matrix-vector produkt berekening, het kan zeer lonend zijn om toch een algoritme als geheel in de vectorisatie-analyse te betrekken.

Voor een matrix A, gerepresenteerd door de elementen A_{ij}, i = 1,..., M en j = 1,..., N, en een vector X, gerepresenteerd door de elementen X_i, i = 1,...., N, wordt het matrix-vectorprodukt gedefiniëerd als:

$$Y_i = \sum_{j=1}^{N} A_{ij}X_j,\ i = 1,....,M \qquad \text{(MV-SDOT versie).}$$

We gaan nu weer verder uit van verwerking binnen een FORTRAN omgeving. In

FORTRAN wordt een matrix kolomsgewijs opgeslagen, dat wil zeggen: $A_{1j}, A_{2j}, \ldots, A_{Mj}$ (de j-de kolom van A) vormen een aaneengesloten vector (stride = 1). Bij de snelheidsanalyse van het algoritme in een taal waarin een rijgewijze opslagstructuur gehanteerd wordt, dient men de hier gevolgde benaderingswijze dienovereenkomstig aan te passen.

Op het eerste gezicht levert de analyse van het matrix-vectorprodukt, verder aangeduid met MV, weinig nieuwe gezichtspunten op. De voor het i-de element van Y benodigde matrixelementen van A: $A_{i1}, A_{i2}, \ldots, A_{iN}$ (de i-de rij van A), vormen een vector die evenwel vanwege de FORTRAN opslagstructuur met stride M in het geheugen ligt opgeborgen. De som $\sum_{j=1}^{N} A_{ij}X_j$ is dus niets anders dan het inprodukt (SDOT) van de i-de rij van A met de vector X en vandaar de aanduiding MV-SDOT versie.

We hebben de SDOT operatie uitvoerig in de vorige paragraaf op de korrel genomen en daarmee zou de kous dus af kunnen zijn. Uit de analyse aldaar kunnen we concluderen dat het MV-algoritme een snelheid haalt overeenkomstig die van SDOT en deze SDOT haalde op de meeste machines minstens iets in de buurt van de halve maximumsnelheid. Dat lijkt inderdaad bevredigend.

Hoewel, de CYBER 205 accepteert voor vectoroperaties slechts aaneengesloten vectoren (stride = 1) en een rij van A is dat dus niet (tenzij M = 1, maar dat is flauw). Dus op een CYBER 205 gaat het vectorfeestje eventjes niet door en wordt het MV-schema uitgevoerd met scalaire snelheid: 5 à 6 Mflops. Op een paar andere machines, zoals de FUJITSU VP-200 en de CONVEX C-1, wordt een stride = 1 ook niet erg geappreciëerd en is vaak een snelheidsverlies in de orde van 50% te verwachten. Bovendien was op de meeste machines, met name de CRAY X-MP, de waarde van $n_{\frac{1}{2}}$ voor SDOT wel wat aan de hoge kant, zodat N groot moet zijn om van de lonkende hoge snelheden te genieten. Als dan bijvoorbeeld ook nog M toevallig gelijk is aan een geheel veelvoud van het aantal geheugenbanks (zie hoofdstuk 4), dan liggen de elementen van een rij steeds in dezelfde bank zodat we een behoorlijke vertraging ten gevolge van memory bank conflicten kunnen verwachten. Kortom, we zijn nu voldoende gemotiveerd om het schijnbaar zo onschuldige matrix-vectorprodukt eens wat verder te analyseren. In een schema voluit geschreven ziet het MV-algoritme er als volgt uit:

$$
\begin{aligned}
Y_1 &= A_{11}X_1 + A_{12}X_2 + \ldots\ldots + A_{1N}X_N \\
Y_2 &= A_{21}X_1 + A_{22}X_2 + \ldots\ldots + A_{2N}X_N \\
&\vdots \\
Y_M &= A_{M1}X_1 + A_{M2}X_2 + \ldots\ldots + A_{MN}X_N
\end{aligned}
$$

Als we de j-de kolom van A noteren met $A_{.j}$, dat wil zeggen dat j vast is en dat de eerste index loopt, dan kan het schema dus geschreven worden als:

$$Y = X_1A_{.1} + X_2A_{.2} + X_3A_{.3} + \ldots\ldots + X_NA_{.N}$$

Met andere woorden: de vector Y is een lineaire combinatie van de kolommen van A met als coëfficiënten juist de elementen van X. Men kan het MV dus ook als volgt berekenen:

a. Zet eerst alle Y_j op nul
b. Bereken voor i = 1,...., N (MV-SAXPY versie)
$Y = Y + X_i * A_{.i}$

Het hele algoritme is nu teruggebracht tot N SAXPY's voor keurig aaneengesloten vectoren (stride = 1). Ook de SAXPY hebben we al tegen het licht gehouden en we hebben gezien dat dit op de meeste machines een prettige vectoroperatie is, wat wil zeggen dat er een meestal hoge R_∞ en een vrij lage $n_{\frac{1}{2}}$ is.

Voor een hoge snelheid van de SAXPY is wel een goede verbinding met het centrale geheugen vereist, bijvoorbeeld 2 load en 1 store unit zoals bij de CRAY X-MP en de CYBER 205. Bij de CRAY-1 en ook bij de CRAY-2, lopen we tegen het beperkte load- en store mechanisme op en loopt de berekening van het MV via SAXPY, voor grote N, zelfs aanmerkelijk trager dan via SDOT. Bijvoorbeeld voor N = M = 300 haalt de SDOT versie op de CRAY-1 een snelheid van 57 Mflops, terwijl de SAXPY versie achterblijft met slechts 41 Mflops.

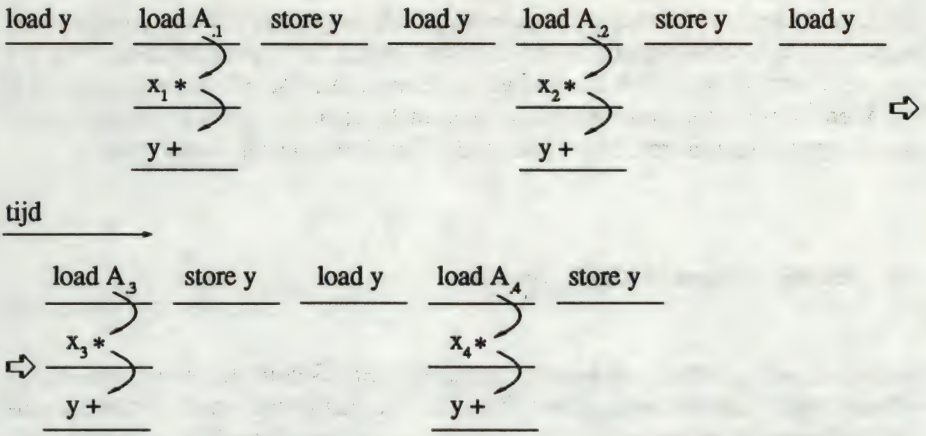
Een, in principe vermijdbare, bottleneck voor de vectorregister-machines is dat voor elke SAXPY de vector Y eerst geladen moet worden en na bijwerking weer weggeschreven. Bij de volgende SAXPY moet dan de Y die zojuist was weggeschreven weer opgehaald worden. Er zijn momenteel geen compilers, voor zover ons bekend althans, die zo slim zijn dat ze dat overbodige wegschrijven en weer ophalen van de 'tussenversies' van Y weten te vermijden. Een probleem daarbij is namelijk dat de optredende vectoren meestal wel langer zullen zijn dan de vectorregisterlengte, zodat het gehele tussenresultaat niet in het register gehouden kan worden. Na afloop van een SAXPY staat alleen het laatste partje van de zojuist bijgewerkte Y nog in het register.

Gelukkig kunnen we de machine een handje helpen. We verdelen daartoe, zoals gesuggereerd in [12], de kolommen van A in groepjes van 4 (bijvoorbeeld). Telkens als we een Y 'binnen' hebben tellen we er in plaats van 1 kolom maar meteen 4 kolommen bij op alvorens Y weer te moeten wegschrijven. Als we voor het gemak maar aannemen dat N een viervoud is dan ziet het algoritme er als volgt uit:

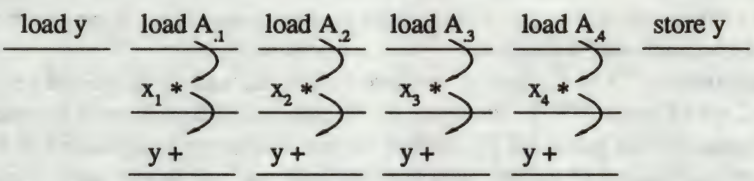
a. zet de Y_i eerst op nul;
b. bereken voor i = 1, 5, 9,...., N - 3 (MV-SAXPY versie)
$Y = Y + X_i * A_{.i} + X_{i+1} * A_{.i+1} +$
$X_{i+2} * A_{.i+2} + X_{i+3} * A_{.i+3}$

We hebben dit aangeduid, in de stijl van [12], als SMPY4, dat wil zeggen: *M*eervoudige SA*XPY* operaties in groepen van *4*, de S slaat op *S*ingle precision).

Voor de eerste vier kolommen van A vergelijken we nu de SAXPY-versie met de SMPY4 aanpak, voor de CRAY-1. Eerst het verwerkingsschema voor de SAXPY-versie:



Dus per groep van vier geldt $k_A = 8$ en $k_c = 12$ en de (theoretische) bovengrens voor de snelheid wordt $R_\infty = 8/12 * 80 \approx 53$ Mflops. Nu het SMPY4-schema:



Dit schema leidt met $k_A = 8$ en $k_c = 6$ tot $R_\infty = 8/6 * 80 \approx 107$ Mflops, een verdubbeling ten opzichte van het vorige schema. Uiteraard kan door met nog grotere groepjes te werken de snelheid verder verhoogd worden tot dicht bij de maximale machinesnelheid van 160 Mflops.

Zelfs voor die machines die in principe voldoende toegangsmogelijkheden hebben tot het centrale geheugen werkt deze aanpak zeer heilzaam. Voor de CRAY X-MP levert het snellere code op vanwege een reductie in de stripmining kosten (en ook vanwege een reductie in de memory bank conflicten ten gunste van de toegang die opgeëist wordt door andere gebruikers die op de andere processoren bezig zijn; hierover later meer). Ook op de CYBER 205, de speciale 'linked triad'-machine bij uitstek, treedt een bescheiden snelheidswinst op omdat de opstarttijd gereduceerd kan worden (de berekening van de vertragingen in de vectorstromen (zie hoofdstuk 5) kan wat geoptimaliseerd worden voor een hele groep tegelijk).

Bij de meeste machines hebben de fabrikanten gezorgd voor geoptimaliseerde snelle routines voor de berekening van het MV, geheel volgens de principes van de groepsaanpak, zoals bij SMPY4. Met deze routines kunnen, mits M maar groot genoeg is, snelheden worden gerealiseerd die dicht bij de maximale machinesnelheid liggen. Voor N = M = 300 haalt de subroutine MXV op de CRAY-1 een snelheid van 130 Mflops.

Voor kleine waarden van M levert de SDOT-versie veelal een beter resultaat (als N tenminste niet te klein is). De moraal van dit verhaal is dat men zich in alle gevallen terdege op de hoogte moet stellen van de mogelijkheden die een algoritme biedt om de

speciale architectuureigenschappen uit te buiten. Men moet daarbij de vectorlengten die bij daadwerkelijke berekening zullen optreden terdege in de gaten houden omdat het niet altijd vanzelfsprekend is dat de bijgeleverde software voor alle mogelijke situaties van M en N de optimale keuze doet. Een eigen analyse kan dan steeds de verwachtingen aan de realiteit toetsen zodat men waar nodig van algoritme kan veranderen.

7.4 BLAS - Extended BLAS - libraries

Een tiental jaren geleden heeft men de zogenaamde BLAS-collectie ontworpen (BLAS = Basic Linear Algebra Subprograms) [43]. Het idee daarbij was om een aantal eenvoudige basisoperaties te standaardiseren, zodanig dat daarmee ingewikkelde lineaire algebra operaties (bijvoorbeeld het oplossen van een stelsel vergelijkingen) konden worden beschreven. Door nu op een gegeven machine de BLAS-set in geoptimaliseerde versie beschikbaar te stellen kan men de gebruiker verzekeren van een efficiënte verwerking van ingewikkelder codes. Prettiger leesbaarheid en overdraagbaarheid van programma's werden hierdoor ook bevorderd.

In de paragrafen 7.1 en 7.2 hebben we twee voorbeelden uit de BLAS-set gezien, namelijk de SAXPY en de SDOT. Dit waren beide, zoals alle BLAS-kernen trouwens, operaties op vectoren. In paragraaf 7.3 hebben we een matrix-vectoroperatie bekeken en we hebben geconstateerd dat daar een hogere snelheid mee was te realiseren dan met SAXPY en SDOT alleen. Naar ervaring blijkt dat vectoroperaties in feite te kleinschalig zijn om het volle rendement uit een vectorcomputer te halen. Die mogelijkheid komt pas volledig aan de orde wanneer er op zijn minst een matrix in het geding is. Het zelfde speelt nog in versterkte mate bij parallelle verwerking. Ook dan is louter vector-vector verwerking in het algemeen veel te kleinschalig en zou men liever zelfs matrix-matrix operaties willen hebben. Op deze parallelle verwerking komen we later nog terug. We beperken ons eerst nog even tot vingeroefeningen op vectorcomputers, vingeroefeningen die ons later zeer van pas zullen komen bij het doorgronden en benutten van parallelle computers.

Om aan de behoefte aan grootschaliger modulen te kunnen voldoen, heeft men zich gebogen over de definitie van een nieuwe set basisoperaties op matrix-vector niveau, die aan de volgende eisen moeten voldoen:

a. ze moeten voldoende mogelijkheden bieden om efficiënte parallelle- of vectorverwerking mogelijk te maken;
b. belangrijke algoritmen voor lineaire algebraproblemen, zoals oplossen van stelsels, singuliere waardenproblemen en eigenwaardeproblemen, moeten, althans voor de meest rekenintensieve gedeelten, uitgedrukt kunnen worden in deze basisoperaties;
c. de basisoperaties moeten niet leiden tot meer rekenwerk, in termen van flops, en ook mag de rekenprecisie niet nadelig aangetast worden.

Dit heeft geresulteerd in de *Extended BLAS* [13]. Voor verschillende machines heeft men reeds gezorgd voor efficiënte implementaties van deze verzameling routines en evenzo zijn (of worden) belangrijke programmabibliotheken (libraries), zoals IMSL [34], LINPACK [19] en NAG [48], gedeeltelijk opnieuw opgezet in termen van deze Extended BLAS, of ook wel *Level 2 BLAS* genoemd. Het zij voor de goede orde nog benadrukt dat het hier louter een herstructurering van algemeen aanvaarde en geaccepteerde algoritmen betreft, het totaal aantal flops en het afrondfoutenpatroon worden hier niet door beïnvloed. Inmiddels heeft men reeds een aanzet gemaakt tot de *Level 3 BLAS*, die bestaan uit matrix-matrix operaties.

Ook voor de individuele gebruiker is het gebruik van de (Extended) BLAS zeer aan te bevelen. Als men er zorg voor draagt dat de gebruikte BLAS, die in FORTRAN gedefiniëerd zijn, beschikbaar zijn op diverse computers die men gebruikt (mini's en pc's) dan draagt dit bij tot in hoge mate overdraagbare codes die dan voor kleine problemen eenvoudig te testen zijn op een kleine computer in de directe werkomgeving. Voor grote problemen kan men hetzelfde programma, of onderdelen daarvan, met behulp van reeds geoptimaliseerde BLAS, zeer efficiënt laten verwerken op supercomputers van uiteenlopende architectuur. Gezien de stormachtige ontwikkelingen op het gebied van architectuur en de grote verschillen daarin, lijkt het niet gewaagd om vast te stellen dat een programma dat eenzijdig op een bepaalde computer is toegesneden bij voorbaat reeds een korte levensduur zal hebben. Het is daarom van groot belang om een te ontwikkelen code zo transparant en overdraagbaar mogelijk te maken (en te houden). Het gebruik van een doorzichtige datastructuur en van goede standaard libraries, die opgezet zijn op basis van verstandig gekozen basisoperaties (BLAS), maken zulks in hoge mate mogelijk.

7.5 IJle matrices

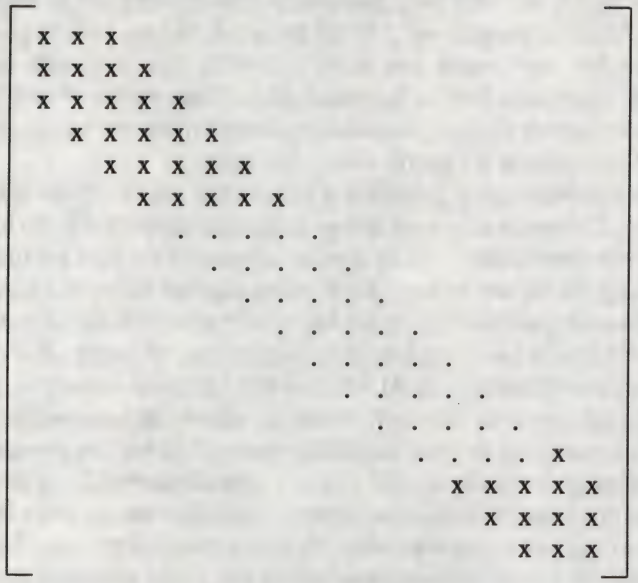
Het komt bij de zeer grote problemen uit de lineaire algebra slechts zelden voor dat de daarin optredende matrices een hoog percentage elementen ongelijk aan nul bevatten. Integendeel, meestal zijn per rij van de matrix slechts een handvol elementen ongelijk aan nul en vaak staan deze elementen ook nog op systematische wijze door de matrix verspreid Men kan daar natuurlijk voordeel mee doen door slechts deze niet-nul elementen op te slaan en bovendien algoritmen te selecteren die zoveel mogelijk de niet-nul structuur van de matrix in tact laten. Voor vectoriële verwerking heeft een en ander tot gevolg dat er een geschikte data opslag gekozen moet worden om bij berekeningen lange vectoren mogelijk te maken. Bovendien kunnen we door de speciale structuur van de matrix niet zonder meer gebruik maken van dezelfde aanpakken als besproken voor de volle matrix-vector vermenigvuldiging in paragraaf 7.3.

We nemen een eenvoudig, maar realistisch voorbeeld. Van de vierkante matrix A staan slechts op de middelste vijf diagonalen elementen ongelijk aan nul (er mogen op deze diagonalen natuurlijk ook best nullen staan, we nemen echter aan dat het er zo weinig zijn dat we er geen systematisch gebruik van kunnen maken). Voor de elementen a_{ij} van A geldt dus:

$a_{ij} = 0$ als $|i - j| > 2$, met $i, j = 1, 2, 3, ..., N$

De niet-nul structuur kunnen we als volgt in beeld brengen:

A =



Dit soort matrices speelt een rol bij het oplossen van bepaalde partiële differentiaalvergelijkingen.

De centrale band van A kunnen we op verschillende manieren opslaan in het geheugen.

a. Rijgewijze opslag

De rijen van A worden, voor zover ze de band overlappen, gewoon achter elkaar opgeborgen in één lang array:

$a_{11}\ a_{12}\ a_{13}\ a_{21}\ a_{22}\ a_{23}\ a_{24}\ a_{31}\ a_{32}\ a_{33}\ a_{34}\ a_{35}\ a_{42}\ a_{43}$ .....

Omdat we weten wat de structuur van A is, kunnen we elk gewenst element van A uit deze brei getallen terugzoeken, Om dit zoekwerk wat te vereenvoudigen kunnen we afspreken om standaard voor *elke* rij 5 getallen op te slaan, dus met name ook voor de eerste twee en de laatste twee rijen van A. Waar nodig vullen we de rijtjes gewoon aan met nullen:

$\underbrace{0\ 0\ a_{11}\ a_{12}\ a_{13}}_{5}$ $\underbrace{0\ a_{21}\ a_{22}\ a_{23}\ a_{24}}_{5}$ $\underbrace{a_{31}\ a_{32}\ .....}_{5}$ ....

Stel dat we deze getallen opbergen in het array AA, dan kunnen we het matrixelement a_{ij} (met $|i - j| \leq 2$, de overige zijn gelijk aan 0) terugvinden in AA $(5(i - 1) + j - i + 3)$.

Nu gaan we eerst maar eens bekijken hoe deze opslagstructuur uitpakt in een concreet rekenvoorbeeld. Een veel voorkomende klus is de berekening van de vector $y = Ax$ voor gegeven vector x. Door alleen rekening te houden met de band van A kunnen de elementen van y geschreven worden als:

$$
\begin{aligned}
y_1 &= a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\
y_2 &= a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 \\
y_3 &= a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5 \\
y_4 &= \quad a_{42}x_2 + a_{43}x_3 + a_{44}x_4 + a_{45}x_5 + a_{46}x_6 \\
y_5 &= \quad\quad a_{53}x_3 + a_{54}x_4 + ... \\
&\vdots
\end{aligned}
$$

De waarde van een element y_i is dus gelijk aan het inprodukt van het bandgedeelte van de i-de rij van A met een segmentje van de vector x. Weliswaar staat het betreffende stukje van de i-de rij op aaneengesloten wijze in het array AA, maar het is natuurlijk uitgesloten dat we op deze manier ook maar enigszins op snelheid komen bij vectorverwerking gezien de wel erg korte lengte van de optredende partjes.

De bij de volle matrix-vectorvermenigvuldiging gebruikte truc om de kolommen van A te accumuleren in y werkt ook niet omdat de (kleine) bandgedeelten van de kolommmen elkaar maar nauwelijks overlappen. Het idee zet ons echter wel op het spoor van diagonaalsgewijze accumulatie. Daartoe schrijven we de diagonalen van A als vectoren ter lengte n (de eventueel ontbrekende elementen vullen we weer gewoon aan met nullen). De onderste diagonaal wordt dan:

$$\tilde{a}_{-2} = (a_{1,-1}, a_{2,0}, a_{3,1}, a_{4,2}, a_{5,3}, \ldots\ldots, a_{n,n-2})$$

De in werkelijkheid niet bestaande elementen $a_{1,-1}$ en $a_{2,0}$ zijn hier toegevoegd om straks in het algoritme een eenvoudiger structuur te krijgen, deze twee elementen krijgen de waarde 0. De overige vier diagonalen worden overeenkomstig geschreven als:

$$
\begin{aligned}
\tilde{a}_{-1} &= (a_{1,0}, a_{2,1}, a_{3,2}, a_{4,3}, \ldots\ldots, a_{n,n-1}) \\
\tilde{a}_{0} &= (a_{1,1}, a_{2,2}, a_{3,3}, a_{4,4}, \ldots\ldots, a_{n,n}) \\
\tilde{a}_{1} &= (a_{1,2}, a_{2,3}, a_{3,4}, a_{4,5}, \ldots\ldots, a_{n,n+1}) \\
\tilde{a}_{2} &= (a_{1,3}, a_{2,4}, a_{3,5}, a_{4,6}, \ldots\ldots, a_{n,n+2})
\end{aligned}
$$

Het zal voorts ook handig blijken de vector x aan te vullen met de elementen x_{-1}, x_0, x_{n+1} en x_{n+2}. Ook deze elementen krijgen de waarde 0 toegekend. Tenslotte definiëren we nog de volgende 5 aaneengesloten segmenten $\tilde{x}$ ter lengte n uit x:

$$
\begin{aligned}
\tilde{x}_{-2} &= (x_{-1}, x_0, x_1, x_2, \ldots\ldots, x_{n-2}) \\
\tilde{x}_{-1} &= (x_0, x_1, x_2, x_3, \ldots\ldots, x_{n-1})
\end{aligned}
$$

$$\tilde{x}_0 = (x_1, x_2, x_3, x_4, \ldots\ldots, x_n)$$
$$\tilde{x}_1 = (x_2, x_3, x_4, x_5, \ldots\ldots, x_{n+1})$$
$$\tilde{x}_2 = (x_3, x_4, x_5, x_6, \ldots\ldots, x_{n+2})$$

De elementen van $\tilde{a}_j$ en $\tilde{x}_j$ nummeren we gewoon van 1 tot n. De vector x staat slechts een keer in het geheugen en deze 5 segmenten $\tilde{x}_j$ slaan we niet apart op; de $\tilde{x}_j$ gebruiken we slechts om een zeker segment in de vector x aan te kunnen duiden.

Men gaat nu eenvoudig na dat de resultaatvector y als volgt geschreven kan worden:

$$y(i) = \tilde{a}_{-2}(i) * \tilde{x}_{-2}(i) + \tilde{a}_{-1}(i) * \tilde{x}_{-1}(i) + \ldots\ldots + \tilde{a}_2(i) * \tilde{x}_2(i)$$
(voor i = 1,2...., n)

Dit is een vectoroperatie met vectorlengte n en kan dus, ten koste van slechts een paar overtollige flops voor de extra ingevoerde elementen, met hoge rekensnelheid verwerkt worden. Omdat evenwel de vectoren $\tilde{a}_j$ niet op aaneengesloten wijze in het geheugen (dat wil zeggen: in het array AA) liggen, hetgeen naast onoverzichtelijke indexberekeningen vaak ook gereduceerde rekensnelheid met zich meebrengt, besluiten we liever tot diagonaalsgewijze opslag van de band van A.

b. *Diagonaalsgewijze opslag*

We slaan nu de diagonalen $\tilde{a}_j$ op in het array AD(N, 5) en wel zo dat $\tilde{a}_{-2}$ in de eerste kolom van AD terecht komt, $\tilde{a}_{-1}$ in de tweede kolom, enzovoort. Uitgeschreven in de elementen van de array's Y, AD en X ziet de berekening van y = Ax er dan uit als:

```
Y(i) = AD(i,1) * X(i-2) + AD(i,2) * X(i-1) + AD(i,3) * X(i)
     + AD(i,4) * X(i+1) + AD(i,5) * X(i+2),     met i = 1,2,....,n
```

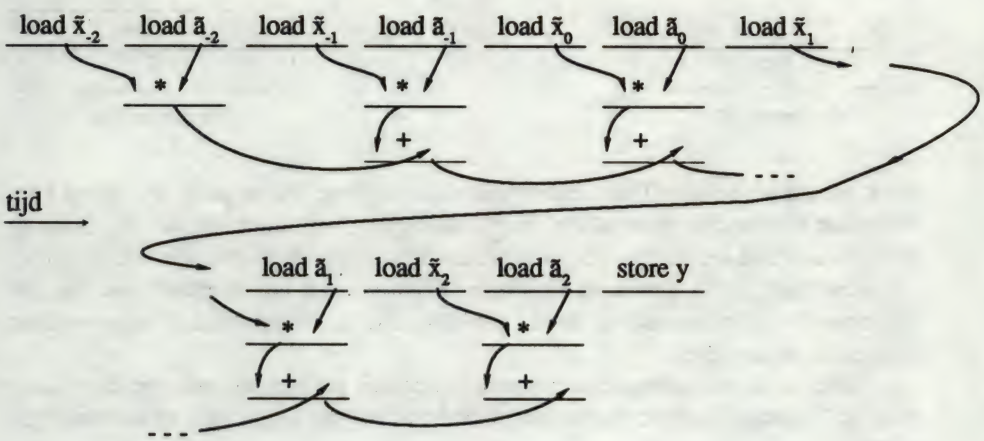
Voor het berekenen van het eerste vectorprodukt AD(i,1) * X(i-2), i = 1,....,n, zijn dus de elementen X(-1), X(0),...., X(n-2) vereist (dat wil zeggen het segment $\tilde{x}_{-2}$). Bij vectorregistermachines wordt dat segment, te beginnen met X(-1), in het register opgeborgen. Voor het tweede produkt, AD(i,2) * X(i-1), is de rij X(0), X(1),...., X(n-1) nodig. Dit segment moeten we weer opnieuw, te beginnen met X(0), in een register laden omdat we nu eenmaal niet kunnen aangeven dat we een register liever vanaf het tweede registerelement zouden willen gebruiken. Een vectorregister kan slechts blindelings als geheel naar een functional unit gevoerd worden. We zouden trouwens ook in de problemen verzeild zijn omdat we dan te maken zouden hebben met registergedeelten van verschillende lengte voor één enkele operatie. Voor de 5 vectorprodukten moeten we dus 5 keer een opeenvolgend segment van x in de vectorregisters laten laden (de compiler doet zoiets automatisch voor ons). Voor direct access machines speelt de mogelijkheid van meervoudig gebruik van de 'x-stroom' al helemaal niet, omdat we één stroom operanden niet twee keer bij dezelfde functional unit kunnen laten arriveren.

We hebben in dit voorbeeld de invloed gezien van de datastructuur op de verwer-

kingssnelheid. Op conventionele scalaire machines had rijgewijze (of kolomsgewijze) opslag zekere voordelen bij het werken met (smalle) bandmatrices. Algoritmen daarvoor, zoals bijvoorbeeld in het pakket LINPACK [19], zijn dan ook op deze opslagstructuur georiënteerd. Waarschijnlijk daarom heeft men bij het ontwerp van de Level 2 BLAS gekozen voor rijgewijze opslag van bandmatrices en het moge uit het voorafgaande duidelijk zijn dat ons dat geen gelukkige keuze lijkt. Wij zullen hier verder de diagonaalsgewijze opslag plus dito verwerking bespreken voor een paar vectormachines. De geïnteresseerde lezer kan dan zelf wel nagaan hoe een en ander uitpakt op een hier niet besproken vectorcomputer.

1. CRAY-1

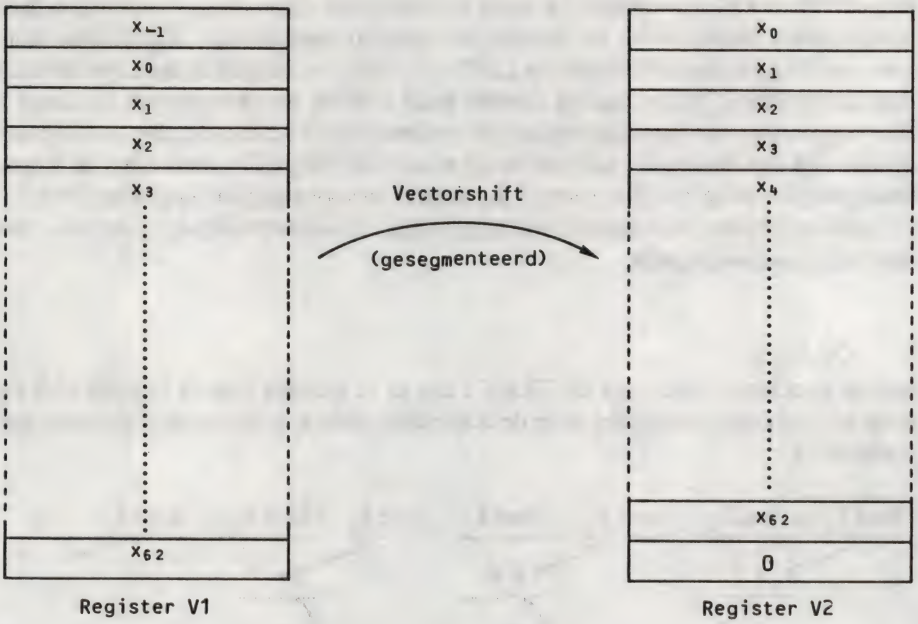
Het verwerkingsschema voor de CRAY-1 ziet er in principe (vanuit FORTRAN) als volgt uit (we gebruiken hierbij weer de diagonaalnotatie $\tilde{a}_j$ en geven de segmenten van x aan met $\tilde{x}_j$):



Figuur 10. Verwerkingsschema voor y = Ax met 5-diagonale A, op de CRAY-1.

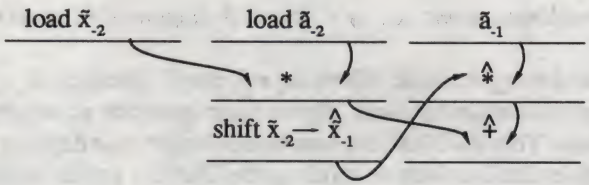
We zien uit figuur 10 dat er gemiddeld 9 flops ($k_A = 9$) per 11 klokcycli ($k_c = 11$) worden verwerkt en dus is de maximum snelheid, afgezien van opstarten en stripmining, circa $9/11 * 80 \approx 65$ Mflops. Voor n = 2000, een bescheiden lengte voor dit soort problemen, meten we 58 Mflops en dat is na wat we nu al van de CRAY-1 gezien hebben, redelijk volgens de verwachtingen.

Het kan, als men daar de nodige moeite in wil steken, allemaal nog wat beter. Er is namelijk de mogelijkheid om, als men in CRAY Assembler Language (CAL) programmeert, gebruik te maken van een instructie waarmee de elementen in een register 1 plaats verschoven overgeschreven kunnen worden naar een ander vectorregister [5]:



Merk op dat de eerste 63 elementen van vectorregister V2 nu juist het eerste part, weliswaar van slechts 63 in plaats van 64 elementen, representeren van $\tilde{x}_{-1}$. Door de inhoud van V2 te vermenigvuldigen met het register waarin de (64) elementen van $\tilde{a}_{-1}$ zitten ontstaan er 63 gewenste tussenprodukten en een nul in plaats van het 64e tussenprodukt. Aan het eind van de rit kunnen we al deze '64e produkten' nog even apart berekenen en aanvullen.

Doordat de gesegmenteerde vectorshift operatie gekoppeld kan worden met de + en/of de * operaties krijgen we het volgende beeld aan het begin van het verwerkingsschema:



Met ^ geven we aan dat slechts plukjes van 63 elementen na de betreffende operatie 'in orde zijn'; om de 64 elementen moet er iets gecorrigeerd worden. Deze truc kan, uitsluitend bij programmeren in CAL, voortgezet worden waardoor er een flink aantal klokcycli kan worden uitgespaard. Wanneer vervolgens de matrix ook nog symmetrisch is, zodat $\tilde{a}_{-2}(i) = \tilde{a}_2(i-2)$ en $\tilde{a}_{-1}(i) = \tilde{a}_1(i-1)$, dan kan ook nog op het laden van de diagonalen worden uitgespaard. Deze kunnen dan gereconstrueerd worden via vectorshifts die gekoppeld worden met de (*, +) cyclus in het schema. Jordan [36] heeft precies aange-

geven hoe op deze wijze verwerkingssnelheden tot boven de 100 Mflops op een CRAY-1 te realiseren zijn.

De lezer moet uit het bovenstaande niet de indruk krijgen dat het altijd gewenst of noodzakelijk zou zijn om in CAL te programmeren. Onze uitweiding is slechts bedoeld om te illustreren wat er in principe met een gegeven hardware architectuur haalbaar is. Omdat CAL-routines vanuit een FORTRAN -programma kunnen worden aangeroepen, zou men bij zeer speciale toepassingen kleine geselecteerde kernen (à la BLAS) in CAL kunnen (laten) schrijven. Voor een groot deel is dat overigens door de fabrikant al gebeurd, bijvoorbeeld voor de BLAS-routines, en dan helpt bovenstaand exposé wellicht bij het beter begrijpen van de dan gemeten Mflops-waarden.

Voor de echte liefhebber melden we tenslotte dat ook in FORTRAN de verwerking op de CRAY-1 nog wat verbeterd kan worden door de loop waarmee y(i) berekend wordt uit te voeren met stappen van 2 en dan telkens 2 opeenvolgende y-waarden per loop-doorgang te berekenen. Voor het gemak nemen we maar aan dat N even is:

```
      N1 = N - 1
      DO 10 I = 1,N1,2
         Y(I) = AD(I,1)*X(I-2)+AD(I,2)*X(I-1)
     $         + AD(I,3)*X(I) + AD(I,4)*X(I+1)
     $         + AD(I,5)*X(I+2)
         Y(I+1) = AD(I+1,1)*X(I-1)+AD(I+1,2)*X(I)
                +AD(I+1,3)*X(I+1)+
                AD(I+1,4)*X(I+2)+AD(I+1,5)*X(I+3)
10    CONTINUE
```

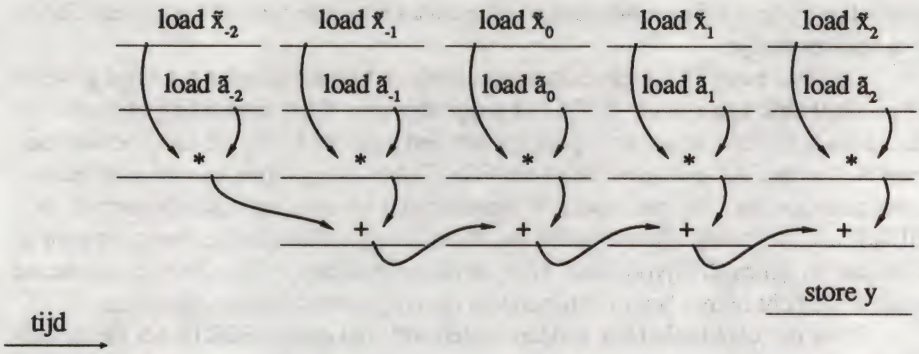
Bovenstaande loop geeft aanleiding tot het laden van de volgende vectoren:

1. (x(-1), x(1), x(3), x(5), x(7), ...)
2. (x(0), x(2), x(4), x(6), x(8), ...)
3. (x(1), x(3), x(5), x(7), x(9), ...)
4. (x(2), x(4), x(6), x(8), x(10), ...)
5. (x(3), x(5), x(7), x(9), x(11), ...)
6. (x(4), x(6), x(8), x(10), x(12), ...)

We zien dat 2 tot en met 5 dubbel gebruikt kunnen worden, waardoor er 4 load-operaties worden uitgespaard. Met wat puzzelwerk kan men nu zelf wel vaststellen dat bovenstaande code aanleiding geeft tot de verwerking van 18 flops per circa 18 klokcycli. Voor een soortgelijk probleem wordt in [58] inderdaad een significant hogere Mflops-waarde gerapporteerd dan voor de orginele DO-loop.

2. CRAY X-MP

De verwerking van de diagonaalsgewijze berekening kan, op 1 processor, schematisch worden weergegeven als:



Figuur 11. Verwerkingsschema voor y = Ax met 5-diagonale A, voor de CRAY X-MP

We zien in een oogopslag dat $k_A = 9$ en $k_c = 5$, hetgeen zou kunnen leiden tot een snelheid van maximaal 9/5 * 105 = 189 Mflops (op 1 CPU met een 9.5 ns klokcyclus). Voor N = 2000 is evenwel slechts een waarde van ≈ 120 Mflops gemeten [39]. We hebben nog niet uitgedokterd waarom de compiler geen kans zag een betere prestatie te leveren. Met de meest recente compiler werd op een 8.5 ns X-MP inmiddels ongeveer 175 Mflops gemeten, zie ook paragraaf 6.3.

Het resultaat van een bewerking op de vectorsegmenten in de registers Vi en Vj wordt altijd eerst in een vectorregister Vk geplaatst met k ongelijk aan i en j, alvorens verder verwerkt of in het geheugen opgeborgen te worden. Dat houdt in dat er voldoende vectorregisters aanwezig moeten zijn om allerlei tussenresultaten vast te houden. Uit analyse van door de X-MP FORTRAN-compiler gegenereerde objectcode blijkt dat in onze 5-diagonale berekening alle 8 vectorregisters gebruikt worden. Om een indruk van de complexiteit van de organisatie hiervan te geven, laten we zien hoe dit gebeurt. De 8 vectorregisters worden genoteerd als V0, V1, V2, ..., V7. In de 5 achtereenvolgende verwerkingsfasen vindt de volgende registertoewijzing plaats:

1.	load $\bar{a}_{-2}$ —> V5		
	load $\bar{x}_{-2}$ —> V4	* —> V1	
2.	load $\bar{a}_{-1}$ —> V3		
	load $\bar{x}_{-1}$ —> V2	* —> V0 + V1	—> V5
3.	load $\bar{a}_0$ —> V7		
	load $\bar{x}_0$ —> V6	* —> V4 + V5	—> V1
4.	load $\bar{a}_1$ —> V3		
	load $\bar{x}_1$ —> V2	* —> V0 + V1	—> V5
5.	load $\bar{a}_2$ —> V7		
	load $\bar{x}_2$ —> V6	* —> V4 + V5	—> V3 —> store y

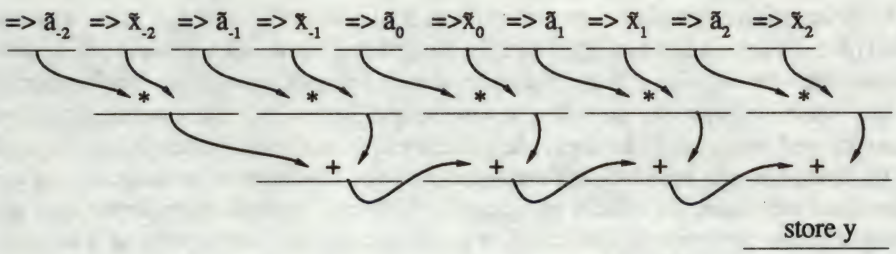
Het toewijzingsschema voor de CRAY-1 is hetzelfde als hierboven. Als we nu loop-unrolling toepassen, zoals aangegeven bij de bespreking voor de CRAY-1, dan zal wel duidelijk zijn dat zoiets alleen helpt wanneer de compiler kans ziet een toewijzingsschema te genereren waarin ook nog plaats is voor het onthouden van te gebruiken vectoren. Deze allerminst triviale opgave wordt inderdaad door de CRAY compilers geklaard. Ook voor de CRAY X-MP leidde dit nog tot wat winst (135 Mflops voor N = 2000), doordat de stripmining overhead wat lager uitviel (minder load operaties!).

3. De overige vectorregistermachines

Het verwerkingsschema voor de CONVEX C-1 verloopt geheel analoog aan dat voor de CRAY-1 (we gaan hier gemakshalve voorbij aan het feit dat de CONVEX nog een vrij groot tussengeheugen heeft, dat bij ophalen en wegbergen van vectoren ook nog een rol kan spelen). We mogen dus een snelheid van maximaal 9/11 * 10 ≈ 8.2 Mflops verwachten. Uit metingen werd een asymptotische top R_∞ = 7.0 Mflops afgeleid; vermoedelijk is het verschil aan stripmining toe te schrijven.

Bij het verwerkingsschema voor de CRAY X-MP zagen we dat alleen in de vijfde fase gebruik gemaakt werd van de drie toegangen tot het centrale geheugen. In de overige fasen werd slechts gebruik gemaakt van twee (load)toegangen. De verwerking op een FUJITSU VP-200 kan dus in principe nagenoeg gelijk uitvallen, slechts het opbergen van het resultaat y moet nog in een aparte fase van N klokcycli plaatsvinden (k_c = 6). De maximale snelheid zou dus 9/6* 267 ≈ 400 Mflops kunnen bedragen. Uit de gemeten snelheden 355 Mflops (voor N = 2000) en 389 Mflops (voor N = 10 000) leiden we af dat R_∞ ≈ 398 Mflops (en $n_{\frac{1}{2}}$ ≈ 245). Het is niet helemaal duidelijk waarom de in [40] gerapporteerde 173 Mflops voor N = 2000 zo drastisch van onze metingen afwijkt (wellicht vanwege een oudere (= minder slimme) compiler).

Op de NEC SX-2 tenslotte, zou vanwege de merkwaardige 'gehalveerde' load fase het verwerkingsschema aanleiding kunnen geven tot k_c = 5.5:



Op dit moment beschikken we slechts over de gemeten waarde 840 Mflops voor N = 2000 [40] en dat komt al een heel eind in de richting van de op grond van k_A = 9, k_c = 5.5 te verwachten maximale top van 9/5.5 * 667 ≈ 1090 Mflops.

4. CYBER 205

Tenslotte beschouwen we nog de CYBER 205, als voorbeeld van een machine die de vectoren direct uit het geheugen haalt. Omdat vectorregisters ontbreken moeten tussenresultaten ook in het geheugen worden opgeborgen. De verwerking van het pentadiagonale matrix-vector produkt zou er dus als volgt kunnen uitzien (met => duiden we het ophalen van vectoren aan en met <= het terugschrijven van een resultaat naar het geheugen):

=> $\tilde{x}_{-1}$	=> t_2	=> $\tilde{x}_0$	=> t_1	=> $\tilde{x}_1$	=> t_1	=> $\tilde{x}_2$	=> t_1
=> $\tilde{a}_{-1}$	=> t_1	=> $\tilde{a}_0$	=> t_2	=> $\tilde{a}_1$	=> t_2	=> $\tilde{a}_2$	=> t_2
*	+	*	+	*	+	*	+
<= t_2	<= t_1	<= t_2	<= t_1	<= t_2	<= t_1	<= t_2	<= y

Met $k_A = 9$ en $k_c = 9$ zou de maximaal haalbare snelheid in principe $k_A/k_c \,.\, 50 \,.\, n_p = 50\, n_p$ Mflops kunnen bedragen (n_p = aantal vector pipes). In [58] worden de volgende gemeten snelheden gerapporteerd: 46 Mflops voor N = 2000, $n_p = 1$ en 88 Mflops voor N = 2000, $n_p = 2$ en dat is redelijk conform de verwachtingen.

We hebben verondersteld dat de compiler zo slim is dat er met de geheugenruimte voor 2 vectoren ter lengte N volstaan kan worden. Bij ingewikkelder vectorexpressies met lange vectoren kan deze extra geheugenruimte voor problemen zorgen, men dient hier bij het reserveren van geheugenruimte op bedacht te zijn.

7.6 Indirecte adressering

Bij belangrijke toepassingen treden matrices op die wel ijl gevuld zijn, maar waarvan het ijlheidspatroon niet zeer regelmatig is. Het is dan gebruikelijk uitsluitend de matrixelementen die ongelijk zijn aan nul op te slaan in het geheugen en het ijlheidspatroon op te slaan via zogenaamde indirecte adressering. Bij indirecte adressering slaat men in een apart array, het index array, de indices van de matrixelementen op die ongelijk zijn aan nul. Er zijn verschillende manieren om dit uit te voeren en het hangt sterk af van het soort toepassing *en* van het type computer (serieel, parallel, vector) welk type opslagstructuur de voorkeur verdient. Een goed overzicht van opbergschema's voor onregelmatige ijle matrices vindt men in [50].

We lichten het werken met indirecte adressering toe aan de hand van een zeer eenvoudig voorbeeld. In paragraaf 7.5 kwamen we een matrix tegen met een fraaie regelmatige structuur. Deze structuur werd nog verder aan het werken met vectorcomputers aangepast door waar nodig de 'niet-nul diagonalen' met nullen op te vullen tot vectoren van gelijke lengte. Als variant hierop bekijken we een matrix waarbij de 5 niet-nul elementen per rij (als het er minder zijn, dan vullen we het aantal gewoon op met voldoende nullen) op onregelmatige wijze over de rij verspreid staan. Nu hangt het sterk

af van het soort algoritme dat we wensen te gebruiken (waarover straks wat meer) hoe we zo'n ijlheidspatroon wensen op te slaan, echter bij verschillende iteratieve methoden kan men volstaan met bewerkingen van het type y = Ax voor gegeven x. In zo'n geval voldoet het volgende opbergschema.

In een array AA(i,j), i = 1,...., N, j = 1,...., 5, slaan we de niet-nul elementen van de matrix A op en wel zo dat AA(i,j) de waarde van het j-de niet-nul element op de i-de rij van A bevat. Bovendien slaan we in een apart index array ind(i,j) de bijbehorende kolomindices van de elementen van A op, zo dat ind(i,j) aangeeft in welke kolom van A het element AA(i,j) op de i-de rij van A thuishoort: A(i, ind(i,j)) = AA(i,j). De elementen van A staan nu indirect geadresseerd opgeborgen in AA. Men gaat eenvoudig na dat voor de berekening van de vector y = Ax het volgende rekenschema voldoet:

$$y(i) = A(i,ind(i,1)) * x(ind(i,1)) + ... + A(i,ind(i,5)) * x(ind(i,5)),\ i = 1,....,N.$$

Dit gaat door de gekozen opslag van de elementen van A over in:

$$y(i) = AA(i,1) * x(ind(i,1)) + ... + AA(i,5) * x(ind(i,5)),\ i = 1,....,N.$$

De hier gekozen wijze van opslag voor A heeft er nu toe geleid dat in bovenstaand schema de elementen y(i), i = 1,....,N, AA(i,1), i = 1,....,N,..., AA(i,5), i = 1,....,N, fatsoenlijke vectoren vormen, echter wat te denken van de rij x(ind(i,1)), i = 1,....,N ?

Het regelmatig toevoeren van de elementen van zo'n indirect geadresseerd x-array wordt in hoge mate belemmerd doordat steeds weer opnieuw eerst in het index adres array gekeken moet worden en daardoor kunnen de memory banks niet op regelmatige wijze worden aangesproken. Indirecte adressering vormt dan ook een probleem voor de meeste supercomputers.

Op de CRAY-1 kan een speciale subroutine GATHER worden gebruikt, die er voor zorgt dat de opeenvolgende elementen x(ind(1,1)), x(ind(2,1)),...., x(ind(N,1)) redelijk snel uit het geheugen worden gehaald en worden opgeborgen in een 'nette' vector $\tilde{x}(1), \tilde{x}(2),...., \tilde{x}(N)$ (dus $\tilde{x}(i) = x(ind(i,1))$). Deze nette vector $\tilde{x}$ kan dan gebruikt worden in de berekening van y voor de berekening van de eerste vectorterm AA(i,1) * $\tilde{x}(i)$. In ons voorbeeld moet men dus voor de berekening van y in totaal vijf keer de GATHER-routine aanroepen om de 5 benodigde versies van x te krijgen.

Het zal duidelijk zijn dat we nu allerminst de rekensnelheden mogen verwachten die we in paragraaf 7.5 tegenkwamen bij de regelmatige ijle matrixstructuur. We zullen dit nader toelichten aan de hand van een tijdmeting die verricht is op een 1-pipe CYBER 205. Op deze machine staat ons de routine Q8VGATH ter beschikking om de indirect geadresseerde vector x(ind(i,1)) te verzamelen in de hulpvector $\tilde{x}$. We bekijken nu drie verschillende situaties, waarbij we slechts 1 vector-vectorprodukt in het rechterlid voor y uitrekenen.

1. Zonder indirecte adressering (de ideale situatie zoals in paragraaf 7.5):

```
      DO 10 I = 1,2000
10    Y(I) = AA(i,1) * X(i)
```

De gemeten CPU-tijd bedroeg $0.492_{10^{-3}}$ sec, dus ~41 Mflops (in dit geval zou $R_\infty \approx 50$ Mflops moeten zijn).

2. Met indirecte adressering (evenwel zonder de creatie van een nette vector):

```
      DO 10 I = 1,2000
10    Y(I) = AA(I,1) * X(ind(I,1))
```

De gemeten tijd bedroeg $0.554_{10^{-2}}$ sec. hetgeen overeenkomt met ongeveer 3.6 Mflops. Omdat de CYBER 205 voor vectoroperaties keurig aaneengesloten vectoren verlangt, hadden we inderdaad dit soort scalaire verwerkingssnelheid mogen verwachten.

3. Met indirecte adressering, nu met de creatie van een nette vector:

```
CALL Q8VGATHR (.....)
```

(We nemen aan dat de aanroep zodanig is dat van de input vector x, met index array ind, de gewenste vector $\tilde{x}$, met $\tilde{x}(i) = x(ind(i))$, geleverd wordt; de precieze aanroep doet hier niet ter zake.)

```
      DO 10 I = 1,2000
10    Y(I) = AA(I,1) * X(I)
```

De gemeten tijd (inclusief de verwerking van Q8VGATHR en de DO-loop) bedroeg nu $0.22_{10^{-3}}$ sec, hetgeen een snelheid van 9.1 Mflops inhoudt.

Uit dit eenvoudige voorbeeld zien we dat indirecte adressering duur kan zijn: in bovenstaand geval ruim vier keer zo duur als directe adressering. Om misverstanden te voorkomen moet hier wel bij aangetekend worden dat er voor de CYBER 205 vele mogelijkheden bestaan om de gevolgen van indirecte adressering te beperken, zodat de schade in voorkomende gevallen beduidend kleiner kan uitvallen als door ons voorbeeld wordt gesuggereerd.

Een aantal fabrikanten heeft inmiddels voor speciale (hardware) voorzieningen gezorgd waardoor indirecte adressering door de compiler in vectorinstructies kan worden omgezet die tot snelheden leiden, een supercomputer waardig. Dit is met name het geval voor de CRAY X-MP, CRAY-2, CONVEX C-1, FUJITSU VP-200 en de HITACHI S810/20.

Volgens bevindingen in [14] zouden veel voorkomende lineaire algebra constructies voor bovengenoemde computers door indirecte adressering slechts ongeveer een factor 2 langzamer worden. Dit stemt overeen met onze eigen experimenten, uitgevoerd op een CONVEX C-1, een CRAY X-MP/2 en een IBM 3090/VF.

Veel problemen uit de toegepaste wetenschappen geven aanleiding tot grote lineaire stelsels vergelijkingen waarvan de matrix ijl gevuld is. Indien men dit soort stelsels te lijf gaat met iteratieve algoritmen (hetzij voor het oplossen, hetzij voor eigenwaardenbepaling, tijdsintegratie of wat dies meer zij), dan kan vaak gewerkt worden met vectoren van behoorlijke lengte, al dan niet indirect geadresseerd. Bij directe methoden,

zoals bijvoorbeeld gauss-eliminatie, ligt dat geheel anders. In dat geval speelt het aantal elementen per rij dat ongelijk is aan nul een rol en dat aantal is in praktische gevallen meestal beduidende kleiner dan de $n_{\frac{1}{2}}$-waarde van vele supercomputers voor de betreffende vectoroperaties.

Omdat door indirecte adressering het prestatievermogen dan bovendien nog eens een factor 2 zakt ten opzichte van directe adressering, steken de snelheden die voor directe methoden gemeten worden vaak pover af tegen die voor iteratieve methoden. Voor een goede discussie over de vectorisatie- en parallelliseringsproblemen die kleven aan directe algoritmen, zie [20]. In [20] worden ook opberg- en adresseringsschema's besproken die geschikt zijn om de ten gevolge van indirecte methoden optredende veranderingen in het ijlheidspatroon van de matrix te kunnen bijwerken. Zowel de lage rekensnelheden als de veranderingen in het ijlheidspatroon (dat door directe algoritmische bewerkingen meestal beduidend minder ijl wordt) bij directe methoden leiden er toe dat het vaker aantrekkelijk wordt om over te stappen op geschikte iteratieve methoden.

ACHT

PROBLEMEN BIJ HET VECTORISEREN

In hoofdstuk 7 hebben we problemen bekeken waarbij de meeste stappen in de algoritmen direct aanleiding gaven tot eenvoudige vectoroperaties. Daarmee is er wel een heel gelukkige probleemselectie gemaakt, want in het algemeen zal men vermoedelijk vaker tegen situaties aanlopen die niet of niet zo eenvoudig te vectoriseren zijn.

Omdat vectorisatie pas rendabel is (en mogelijk is) wanneer er voldoende aanbod is van gelijksoortige opeenvolgende eenvoudige operaties, komen slechts die constructies in een programma in aanmerking voor vectorisatie die te schrijven zijn als (eenvoudige) DO-loops. Dat wil allerminst zeggen dat elke eenvoudige DO-loop aanleiding zou kunnen geven tot vectorinstructies. Bovendien is namelijk nog vereist dat de opdrachten op uniforme wijze, onafhankelijk van elkaar, kunnen worden uitgevoerd. Uit de laatste eis vloeit voort dat de eindresultaten van een (basis-)loop als vector beschikbaar moeten komen. We zullen hier een paar speciale loops van commentaar voorzien. Een goede en ook overzichtelijke behandeling van verschillende DO-loops vindt men bijvoorbeeld in een speciale CRAY-handleiding [55]. Ook in FORTRAN-handleidingen en optimalisatiehandleidingen voor andere computers kan men vele nuttige wenken aantreffen (zie bijvoorbeeld ook [27]).

8.1 Enkele probleemgevallen

In deze paragraaf bespreken we enkele vectorisatieproblemen aan de hand van eenvoudige voorbeelden. We beginnen maar met een vectoriseerbare situatie:

a.
```
      DO 10 I = 1,N
      A(I) = B(I+3) * A(I) + C(I) ** 2
      D(I) = A(I) - E * B(I) * C(I)
10    CONTINUE
```

In dit voorbeeld hangt de berekening van de waarden van D(I) weliswaar af van de zojuist berekende A(I), echter deze loop is zonder problemen te splitsen in twee vectoriseerbare loops die samen hetzelfde resultaat als hierboven opleveren:

```
      DO 101 I = 1,N
101   A(I) = B(I+3) * A(I) + C(I) ** 2
      DO 102 I = 1,N
102   D(I) = A(I) - E * B(I) * C(I)
```

Het is evenwel verstandig om deze splitsing slechts in gedachten uit te voeren en *niet expliciet* in het programma zelf door te voeren. Combinatie van loops leidt op conventionele computers vaak al tot efficiëntere code (minder loop-overhead, sommige variabelen kunnen in de registers gehouden worden) en vectorcomputers splitsen een dergelijke loop automatisch op in (meestal) efficiënte vectoroperaties. Door eigenhandig loop-distributie uit te voeren ontneemt men de compiler vele mogelijkheden tot optimalisatie.

Bij vectorregistermachines moeten de meeste compilers in staat geacht worden te detecteren dat voor de berekening van D(I) de vectoren A(I) en C(I) niet opnieuw geladen hoeven te worden (voor B(I) geldt dat natuurlijk niet, zoals we inmiddels weten). Dit is minder triviaal dan het lijkt. Wanneer we uitgaan van een N die groter is dan de vectorregisterlengte K dan worden uiteraard de vectoroperaties uitgevoerd op vectorsegmenten ter lengte K. Zouden we de DO 10-loop klakkeloos vervangen door de beide uitgesplitste loops, dan zijn na afloop van de DO 101-loop nog slechts de laatst verwerkte segmenten van de vectoren A en C in de vectorregisters aanwezig. Voor de berekening van D in de DO 102-loop moeten dan toch weer op zijn minst alle overige segmenten van A en C opnieuw geladen worden. Het zou dus veel slimmer geweest zijn om de DO 20-loop eerst in porties ter lengte K op te delen en pas daarna voor iedere portie loopdistributie toe te passen. Dit is nu precies wat de compilers achter de schermen automatisch voor ons uitvoeren. We doen er dus beter aan onze codes overzichtelijk te houden en voor dit soort optimalisatie te vertrouwen op de compilers.

Overigens hoeft men omtrent het gevectoriseerd uitvoeren van loops niet in het ongewisse te blijven. Alle compilers geven desgewenst aan voor welke loops in een programma gevectoriseerde code wordt gegenereerd en we kunnen dus alle aandacht in eerste instantie wijden aan die loops die niet tot vectorcode leiden. Overigens zij hier opgemerkt dat er situaties zijn waarin de compiler meldt vectorcode te genereren maar waarvoor betere alternatieven denkbaar zijn. De behandeling van deze situaties vereist doorgaans veel kennis en ervaring en valt buiten het bestek van dit boek.

Een voorbeeld van een niet zonder meer vectoriseerbare loop zijn we al tegengekomen, namelijk bij de inprodukt berekening in paragraaf 7.2. We hebben aldaar gezien hoe deze constructie toch grotendeels in vectorcode kon worden uitgevoerd.

In het algemeen wordt vectorisatie van een DO-loop gehinderd wanneer, door welke oorzaak dan ook, een regelmatige en onafhankelijke uitvoering van de verschillende doorgangen van de loop niet plaats kan vinden:

b.
```
      DO 10 I = 1,N
      A(I) = B(I) * C(I+1) -3.0 * D(I+K)
      IF (A(I) . LT . 1.0) GOTO 20
10    CONTINUE
      ........
20    ........
      ........
```

Vectorisatie wordt hier verhinderd omdat een regelmatige onafhankelijke voortgang niet gewaarborgd wordt. Zodra immers een A(I) kleiner is dan 1, dan is de berekening van de daaropvolgende elementen van A, die bij vectorisatie al gedeeltelijk in de pipeline zouden kunnen zitten, zelfs ongewenst.

c.
```
      DO 10 I = 1,N
      CALL ABC(A(I), B(I), C)
10    CONTINUE
      ........
```

met
```
      SUBROUTINE ABC(P,Q,R)
      P = P + R * Q
      RETURN
      END
```

In het algemeen zijn DO-loops waarin een subroutine of function wordt aangeroepen niet vectoriseerbaar, ook niet als de betreffende routine na 'invulling' tot vectorcode aanleiding zou geven. De huidige compilers zijn nu eenmaal niet in staat te voorzien welke neveneffecten een CALL van een externe routine kan hebben. In sommige gevallen, zoals hier, kan vectorisatie alsnog worden afgedwongen door de DO-loop 'binnen de routine te brengen':

```
      CALL ABC1(N,A,B,C)
      .....
```

met
```
      SUBROUTINE ABC1(N,P,Q,R)
      DIMENSION P(N), Q(N)
      DO 10 I = 1,N
10    P(I) = P(I) + R * Q(I)
      RETURN
      END
```

Voor een aantal bekende standaardfuncties wordt deze zogenaamde loop pushing

automatisch door de compiler verzorgd. In zo'n geval heeft de compiler dan de beschikking over de vectorvarianten van de betreffende functies. Dus

```
      DO 10 I = 1,N
10    A(I) = EXP(B(I))
```

wordt door de meeste compilers automatisch vervangen door iets als

```
      CALL EXPV(N,B,A)
```

met hetzelfde resultaat als de DO-loop. Bij twijfel kan men dit weer verifiëren doordat de compiler melding maakt van het feit of de betreffende DO-loop kon worden gevectoriseerd.

d. Voor uitvoer-statements zijn dit soort vectorvarianten meestal niet voorhanden, zodat men er niet op mag rekenen dat DO-loops die I/O bevatten tot vectorcode leiden. In voorkomende gevallen is het daarom raadzaam om I/O zoveel mogelijk in aparte loops onder te brengen. De constructie

```
      DO 10 I = 1,N
      A(I) = EXP(B(I)) + C(I)
      PRINT 20, A(I)
10    CONTINUE
20    FORMAT (.....)
```

kan men dus beter vervangen door

```
      DO 10 I = 1,N
 10   A(I) = EXP(B(I)) + C(I)
      DO 101 I = 1,N
101   PRINT 20, A(I)
 20   FORMAT (.....)
```

In dit geval is tenminste de DO-loop nog vectoriseerbaar. Nog beter is het de DO 101-loop te vervangen door

```
      PRINT 20, A          (als alle elementen van A
                            afgedrukt moeten worden)
```

of

```
      PRINT 20, (A(I), I = 1,N)
```

e. Het voorkomen van conditional statements geeft ook aanleiding tot problemen. Het volgenden voorbeeld is ontleend aan de optimalisatie–handleiding voor de CRAY-1 [55].

```
      DO 10 I = 1,N
      IF (B(I).GT.C(I)) A(I) = A(I) * B(I)
10    CONTINUE
```

Sommige compilers vectoriseren dit soort constructies automatisch, alhoewel meestal niet op pieksnelheden gerekend mag worden. Voor de meeste vectorcomputers zijn echter meerdere hulpmiddelen beschikbaar om dit soort if-statements alsnog om te zetten in een relatief snelle code. Voor de CRAY-1 bijvoorbeeld, kan de bovenstaande DO-loop worden vervangen door de statement

```
      DO 10 I = 1,N
      A(I) = CVMGT(A(I)*B(I),A(I),B(I).GT.C(I))
10    CONTINUE
```

Dit levert voor N = 100 een versnelling op met ongeveer een factor 10 ten opzichte van de originele loop [55]. Met name voor de CYBER 205 en de FUJITSU VP-computers zijn er uitgebreide mogelijkheden om DO-loops met conditional statements te optimaliseren. De juiste keuze hieruit wordt bepaald door het percentage van de gevallen waarin aan de conditie wordt voldaan (de zogenaamde TRUE-ratio). Bij de FUJITSU VP-compilers wordt standaard een keuze gemaakt op grond van statistische gegevens, men kan deze keuze beïnvloeden door in een comment instructie(* VOCL .....) een schatting voor de TRUE-ratio mee te geven (als men daar tenminste een idee van heeft). Voor meer informatie hierover zie [39,56].

f. Dubbele DO-loops (geneste DO-loops):

```
      DO 10 J = 1,N
          DO 10 I = 1,K
10            A(I,J) = B(I,J) + C(I,J)
```

Tenzij K heel klein is, verdient het zonder meer de voorkeur om er voor te zorgen dat de rij-index I als binnenste loop parameter optreedt, vanwege de kolomsgewijze opslagstructuur in FORTRAN. Een enkele FORTRAN-compiler verzorgt deze omwisseling van loops, indien nodig, zelf (de VP-200 compiler).

In het algemeen is slechts de binnenste loop in een geneste structuur kandidaat voor vectorisatie. Als evenwel de indexgrenzen samenvallen met de gedeclareerde array grenzen (en als de compiler dat kan zien, doordat N en K tijdens compilatie bekend zijn), dan genereren sommige FORTRAN-compilers een vectorinstructie die overeenkomt met:

```
      DO 10 I = 1,N*K
10        A(I) = B(I) + C(I)
```

Het grote voordeel hierbij is dat we nu slechts één lange loop hebben in plaats van N kortere loops met N keer opstart overhead.

g. (Schijnbare) recursies. Als voorbeeld bekijken we een DO-loop van de vorm:

```
      DO 10 I = N1, N2
10    A(I) = A(I) + B(I) * A(I+K)
```

Omdat de compiler meestal niet kan zien welke waarde K tijdens executie zal hebben, wordt een dergelijke opdracht niet gevectoriseerd. Als de programmeur echter weet dat K altijd positief zal zijn (of als de compiler dat kan detecteren, bijvoorbeeld als er vlak voor de loop staat K = 3) en als N2 > N1 (voor N1 < N2 moet K juist negatief zijn), dan is A(I) dus niet van *voorgaande* berekeningen (dat wil zeggen: voor vorige I's) afhankelijk en zou er zonder gevaar gevectoriseerde code opgeleverd kunnen worden. Bij de meeste vectorcomputers kan men zulke wetenschap in een zogenaamde comment line meegeven.

Voor de CRAY-computers plaatst men dan onmiddellijk voor de betreffende loop een zogenaamde CDIR$-instructie (beginnend in de eerste positie van de regel, omdat het als comment herkend moet worden):

```
CDIR$ IVDEP
      DO 10 I = N1, N2
 10   A(I) = A(I) + B(I) * A(I+K)
```

(IVDEP wil zeggen: *I*gnore *V*ector *DEP*endency; de andere vectorcomputers kennen soortgelijke Comment directives.)

De compiler genereert nu zonder meer de code om de segmenten [A(N1), A(N1+1),...., A(N2)], [B(N1), B(N1+1),...., B(N2)] en [A(N1+K), A(N1+K+1),...., A(N2+K)] te laden, waarbij *alle* A-elementen de waarden bevatten, zoals ze die voor ingang van de loop hadden. Met deze vectorsegmenten worden dan, op de bekende vectoriële wijze, de waarden voor het segment [A(N1), A(N1+1),...., A(N2)] vervangen door de nieuwe waarden.

Als nu echter per ongeluk in de bovenstaande situatie K de waarde -1 zou hebben gehad, dan wordt door de CDIR$ IVDEP instructie geforceerd dat de A(I)'s berekend worden met de waarden van A(I-1) zoals die *voor* ingang van de loop golden en *niet* met de zojuist berekende nieuwe waarde (zoals de bedoeling was). Bij het gebruik van zo'n comment instructie moet men dus zeker weten dat er bij executie geen recursie *binnen* de loop optreedt.

Hopelijk is het uit deze discussie duidelijk dat het gebruik van zo'n comment directive ook nuttig en zinvol is als, bij N2 > N1, K negatief is, echter zo dat N2 + K < N1. Immers ook dan treedt er geen recursie binnen de loop op.

Zoals reeds eerder opgemerkt, kennen alle vectorcomputer FORTRAN-compilers, uitgezonderd die voor de CYBER 205, dit soort comment directives. Voor de CYBER 205 kan men vectorisatie van dit soort loops, waarbij dus geen recursie binnen de loop optreedt, afdwingen door middel van het gebruik van vector syntax (specifiek voor de CYBER 205). Het voordeel van comment directives boven computerspecifieke statements is evenwel dat daardoor de overdraagbaarheid van het programma naar andere computers (bijvoorbeeld een mini of een pc om het programma voor kleine problemen

te testen) niet in het gedrang komt. Voor de CYBER 205 kan men ook een comment directive gebruiken in combinatie met een speciale vectoriserende precompiler, de zogenaamde VAST compiler van Pacific Sierra Corp. Deze VAST compiler maakt van een gegeven FORTRAN programma een nieuw FORTRAN programma dat zich beter door de CYBER 205 compiler laat vectoriseren. Onder invloed van zo'n comment directive vervangt VAST de hier besproken statements door CDC CYBER 205 vector syntax (Pacific Sierra Corp. levert soortgelijke precompilers die geschikt zijn voor vectorisatie van FORTRAN programma's ook voor andere supercomputers).

8.2 De vectordeling

Tot nu toe hebben we in alle voorbeelden slechts optel- en vermenigvuldigoperaties gebruikt. Dat lijkt een toevalligheid, maar dat is het allerminst. De waarheid is dat vectorcomputers grote moeite hebben met de deling. Hun compilers zullen weliswaar aangeven dat ze DO-loops waarin delingen voorkomen kunnen vectoriseren, maar vergeleken met de vermenigvuldiging zijn de resultaten qua snelheid vrij laag. Dat komt omdat er geen speciale gesegmenteerde functional unit voor de deling beschikbaar is, welke in 1 rekengang de deling in volledige precisie uitvoert (dat wil zeggen met 1 resultaat per klokcyclus), zoals dat wel het geval is voor de vermenigvuldiging en de optelling. Kennelijk laat de deling geen eenvoudige segmentatie toe. We illustreren dit met het volgende simpele voorbeeld:

```
      DO 10 I = 1,N
10    A(I) = B(I)/C(I)
```

Hiervoor hebben we de volgende asymptotische snelheden gemeten (met tussen haakjes de waarde die gemeten wordt voor dezelfde loop met de vermenigvuldiging in plaats van de deling, aangegeven door R (*)):

CYBER 205:	$R_\infty \approx$ 16 Mflops	(R_∞ (*) $\approx$ 100 Mflops)
CONVEX C-1:	$R_\infty \approx$ 1.5 Mflops	(R_∞ (*) $\approx$ 3 Mflops)
CRAY-1:	$R_\infty \approx$ 13.6 Mflops	(R_∞ (*) $\approx$ 22 Mflops)
CRAY X-MP (8.5 ns):	$R_\infty \approx$ 31 Mflops	(R_∞ (*) $\approx$ 80 Mflops)
IBM 3090 (1VF):	$R_\infty \approx$ 2.6 Mflops	(R_∞ (*) $\approx$ 8,3 Mflops)
NEC SX-2:	$R_\infty \approx$ 60 Mflops	(R_∞ (*) $\approx$ 280 Mflops)

Bedenk wel dat voor de CRAY-1 en de CONVEX C-1 het effect van de trage deling nog versluierd wordt doordat er 3 load/store operaties in het spel zijn. We zullen de trage deling nader toelichten.

Omdat de deling dus niet in eenvoudige gesegmenteerde vorm voorhanden is, moesten er andere wegen gezocht worden om nog zoiets als vectorsnelheid te halen. Het algoritme dat hiervoor doorgaans gebruikt wordt is het bekende Newton iteratieproces.

Voor het oplossen van de vergelijking f(x) = 0, met x en f(x) reële getallen, luidt dit proces:

$x^{i+1} = x^i - \frac{f(x^i)}{f'(x^i)}$, met x^i een gegeven startwaarde.

Als men van een reëel getal a de inverse wenst te bepalen, dan kan men het Newton proces uitvoeren op de vergelijking 1/x - a = 0, die inderdaad als oplossing x = 1/a heeft. Dat levert als iteratieproces op

$x^{i+1} = x^i * (2 - x^i * a)$

Men kan bewijzen dat dit proces in weinig slagen de gewenste waarde van 1/a (tot op machineprecisie) oplevert, als men een redelijke benadering voor 1/a als startwaarde neemt.

Door het Newton proces 'simultaan' uit te voeren voor alle vectorelementen C(i) ontstaat vectoriseerbare code. We lichten dit nader toe aan de hand van de verwerking op de CRAY-1. Op deze machine is, evenals op de CYBER 205, een gesegmenteerde functional unit beschikbaar die de waarde van 1/a oplevert in *halve* precisie. Dit tussenresultaat wordt als beginschatting voor het Newton proces gebruikt en men kan dan met slechts 1 iteratieslag volstaan om volle machineprecisie te bereiken.

Als we nu voor het gemak maar even aannemen dat de vectoren B en C reeds in de vectorregisters aanwezig zijn en dat het resultaat niet gestored hoeft te worden, dan ziet de verwerking van de 'pure' deling er schematisch (afgezien van stripmining) als volgt uit:

a. Bereken de inversen van C(i) in halve precisie (we geven het resultaat aan met U(i). Vermenigvuldig U(i) alvast met B(i) (het resultaat geven we aan met V(i)). Deze bewerking kan op de CRAY-1 gekoppeld worden met de halve precisie-deling;
b. Bereken U(i) * C(i) (te noteren als W(i))
   en vervolgens (als gekoppelde operatie)
   2.0 - W(i) (te noteren als Y(i))
c. Tenslotte berekenen we A(i) = Y(i) * V(i)

De pure deling kost dus gemiddeld drie keer zoveel klokcycli als de geïsoleerde vectorvermenigvuldiging, simpelweg omdat het hele proces 3 vectorvermenigvuldigingen omvat, die (uiteraard) niet met elkaar gekoppeld kunnen worden.

Voor de DO 10-loop in ons voorbeeld komen daar voor de CRAY-1 en de CONVEX C-1 in dit geval nog 3 load/store operaties bovenop (die kennelijk niet gekoppeld worden met stappen uit het delingsproces, hetgeen mogelijk zou zijn), wat verklaart dat de deling in dit geval ongeveer twee keer zo traag is als een vermenigvuldiging in een soortgelijke loop (namelijk ≈ 6N in plaats van ≈ 3N klokcycli). De verwerking op de CYBER 205 verschilt in zoverre dat in stap a de deling niet gekoppeld kan worden met de vermeningvuldiging en dat de beide operaties in stap b samengenomen kunnen worden tot een zogenaamde linked triad. Inclusief de load- en store operaties (immers direct memory access) komt de vectordeling dan op ≈ 4N klokcycli tegen ≈ N klokcycli voor

de vectorvermenigvuldiging, hetgeen een snelheidsreductie met een factor 4 verklaart (in werkelijkheid bedraagt de reductie ≈ 6).

De nog weer tragere verwerking op de Japanse supercomputers (op de FUJITSU VP-200 kost de pure deling ongeveer 7 keer zoveel als de pure vermenigvuldiging, dat wil zeggen: afgezien van load/store operaties) duidt op de uitvoering van 2 Newton slagen (speculatieve gedachte).

Het is dus raadzaam om de deling in rekenprocessen zo veel mogelijk te voorkomen. Als er meerdere malen door de vectorelementen C(i) gedeeld moeten worden (zonder dat deze tussentijds van waarde veranderen), dan kan men deze beter eenmalig inverteren en apart opslaan. In plaats van constructies als

```
      DO 10 I = 1,N
10    C(I) = B(I)/S
```

kan men (meestal) beter schrijven

```
      SINV = 1.0/S
      DO 10 I = 1,N
10    C(I) = B(I) * SINV
```

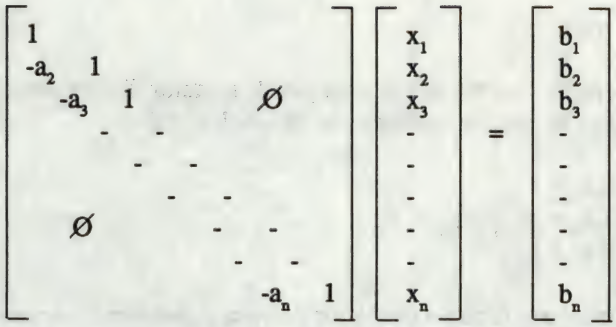
Menige compiler herkent tegenwoordig zelf ook wel dat S een loop invariant is en dat 1/S beter even apart kan worden uitgerekend, men mag daar echter niet zonder meer van uitgaan. Het verschil tussen beide loops op de CONVEX C-1, die verder overigens een goed vectoriserende en optimaliserende compiler bezit, bedraagt ongeveer een factor 2.5 in snelheid en voor de IBM 3090/VF bedraagt het verschil zelfs een factor 5.

8.3 De lineaire recursie (of het oplossen van een bidiagonaal stelsel)

De lineaire recursie op vector en parallelle computers vormt een probleem waaraan reeds zeer veel aandacht is geschonken. Er zijn verschillende technieken bedacht om tot vectoriseerbare algoritmen te geraken. Deze technieken zijn toepasbaar op een veel wijdere klasse van problemen (en daar ook meestal voor ontworpen), maar ze laten zich aardig beschrijven voor de volgende eenvoudige recursie.

$$x_1 = b_1$$
$$x_i = b_i + a_i * x_{i-1},\ i = 2,3,....,n$$

In termen van lineaire algebra is de vector $(x_1,...., x_n)$ te schrijven als de oplossing van het bidiagonale stelsel:



Dit probleem speelt vaak een voorname rol bij het oplossen van bijvoorbeeld gediscretiseerde (partiële) differentiaalvergelijkingen. Het zal een ieder duidelijk zijn dat rechttoe recht-aan code voor dit probleem niets vectoriseerbaars oplevert en de scalaire verwerkingssnelheid leidt op vectorcomputers tot een behoorlijke verlaging van de totale snelheid van het algoritme, waar deze recursie mogelijk slechts een relatief klein gedeelte van uitmaakt (denk aan de wet van Amdahl, paragraaf 6.1). Alle reden dus om uit te zien naar vectoriseerbare alternatieven.

We beschouwen 3 basistechnieken om de recursie te lijf te gaan en haar op te delen in onafhankelijke of kleinere brokken. We betrekken alleen vectoriële aspecten in onze discussie, de parallelle aspecten komen later nog aan de orde.

a. Recursive Doubling

Deze techniek komt er in wezen op neer dat we voor x_{i-1}, in het rechterlid van de uitdrukking voor x_i, de uitdrukking voor x_{i-1} zelf substitueren. Dat heeft het volgende tot resultaat:

$$\begin{aligned} x_i &= b_i + a_i * x_{i-1} = b_i + a_i * (b_{i-1} + a_{i-1} * x_{i-2}) \\ &= b_i + a_i * b_{i-1} + a_i * a_{i-1} * x_{i-2} \\ &= \tilde{b}_i + \tilde{a}_i * x_{i-2} \end{aligned}$$

Met behulp van deze nieuwe recursie kunnen we, uitgaande van x_1, de rij $x_1, x_3, x_5, \ldots$ voortbrengen en met behulp van $x_2 (= b_2 + a_2 * x_1)$, *onafhankelijk* van deze rij, de rij $x_2, x_4, x_6, \ldots$

In feite hebben we het oorspronkelijke probleem, een recursie ter lengte n, vervangen door 2 onafhankelijke recursies, elk ter lengte n/2. De hoeveelheid scalair rekenwerk is hier dus niet door afgenomen, maar er zijn wel extra kosten gemaakt, namelijk voor de berekening van de coëfficiënten $\tilde{b}_i$ en $\tilde{a}_i$. Voor vectorcomputers lijkt deze techniek dus niet zo geslaagd. Recursive doubling kan mogelijk in sommige parallelle omgevingen wel van nut zijn.

b. Cyclische reductie

Deze techniek is nauw verwant aan de voorgaande techniek. De invultruc wordt nu alleen toegepast voor de oneven geïndiceerde variabelen, dus

$$\begin{aligned} x_{2i} &= b_{2i} + a_{2i} * x_{2i-1} \\ &= b_{2i} + a_{2i} * (b_{2i-1} + a_{2i-1} * x_{2i-2}) \\ &= \tilde{b}_{2i} + \tilde{a}_{2i} * x_{2i-2} \end{aligned}$$

Hierdoor houden we een recursie over voor de even geïndiceerde variabelen en is het oorspronkelijke recursieprobleem dus, ten koste van wat invulwerk, gereduceerd tot een probleem van de halve grootte.

Het berekenen van de coëfficiënten $\tilde{b}_{2i}$ en $\tilde{a}_{2i}$ bestaat uit twee vectorhandelingen:

$$\begin{aligned} \tilde{b}_j &= b_j + a_j * b_{j-1} \\ \tilde{a}_j &= a_j * a_{j-1} \end{aligned} \qquad j = 2, 4, 6, 8, \ldots, n$$

Ze kunnen dus versneld (= vectorieel) worden uitgevoerd. Voor het gemak veronderstellen we maar dat n even is.

Na deze reductieslag, die in feite slechts bestaat uit het berekenen van de coëfficiënten $\tilde{a}_j$ en $\tilde{b}_j$, moet de recursie ter lengte n/2 op de even geïndiceerde variabelen worden uitgevoerd. De hoeveelheid scalair werk, het langzame deel, is dus gehalveerd. Tenslotte kunnen de oneven geïndiceerde x-waarden weer met vectoroperaties uit de even geïndiceerde exemplaren berekend worden:

$$x_i = b_i + a_i * x_{i-1},\ i = 3, 5, 7, \ldots, n-1$$

Omdat de oneven geïndiceerde x-waarden slechts afhangen van even geïndiceerde exemplaren kan vectorisatie veilig worden afgedwongen via bijvoorbeeld de in paragraaf 8.1 besproken comment directives.

Met behulp van de in hoofdstuk 6 geïntroduceerde prestatievermogenparameters R_∞ en $n_{\frac{1}{2}}$ kan de effectiviteit van het cyclische reductieproces geanalyseerd worden en kunnen we ons vooraf een indruk verschaffen of toepassing van het proces onder gegeven omstandigheden zinvol is. Voor deze analyse, die hier als voorbeeld van een algoritme analyse wat uitvoeriger besproken zal worden, nemen we gemakshalve aan dat n te schrijven is als een veelvoud van een geschikte macht van 2. Het is niet moeilijk het proces uit te voeren voor andere waarden van n, maar dat zou de beschrijving nodeloos ingewikkelder maken.

De oorspronkelijke recursie kost ongeveer 2n flops. Als we aannemen dat deze recursie met scalaire snelheid van S Mflops wordt uitgevoerd (scalaire snelheid wordt gekenmerkt door het feit dat zij nauwelijks afhangt van de vectorlengte n), dan bedraagt de rekentijd t_0, uitgedrukt in μsec:

$$t_0 = \frac{2n}{S}$$

Bij uitvoering van 1 slag cyclische reductie hebben we met de volgende componenten in het algoritme te maken:

- een recursie voor de even geïndiceerde x-waarden. De rekentijd hiervoor bedraagt dus n/S;
- het berekenen van de coëfficiënten voor de gereduceerde recursie:

$$\tilde{b}_j = b_j + a_j * b_{j-1}$$

$$\tilde{a}_j = a_j * a_{j-1}$$

$j = 2, 4, 6, 8, \ldots, n$ (dus stride 2)

Indien we de asymptotische vectorsnelheid en de halve vectorsnelheidswaarde voor deze reductiestap aangeven met respectievelijk R_∞ (R) en $n_{\frac{1}{2},R}$, dan bedraagt de verwerkingstijd (met $k_A = 3$ en vectorlengte n/2):

$$\frac{3}{R_\infty(R)} \left(\frac{n}{2} + n_{\frac{1}{2},R}\right)$$

- het berekenen van de oneven geïndiceerde x-waarden via de invulstap

$$x_j = b_j + a_j * x_{j-1} \qquad j = 3, 5, 7, \ldots, n-1$$

neemt, uitgedrukt in de prestatievermogenparameters R_∞ (I) en $n_{\frac{1}{2},I}$, de volgende tijd in beslag ($k_A = 2$ en vectorlengte ~n/2):

$$\frac{2}{R_\infty(I)} \left(\frac{n}{2} + n_{\frac{1}{2},I}\right)$$

De totale rekentijd voor het oplossen van de waarden van x met 1 slag cyclische reductie bedraagt dientengevolge:

$$t_1 = \frac{n}{S} + \frac{3}{R_\infty(R)} \left(\frac{n}{2} + n_{\frac{1}{2},R}\right) + \frac{2}{R_\infty(I)} \left(\frac{n}{2} + n_{\frac{1}{2},I}\right)$$

$$= \frac{n}{S} + \frac{n}{S} \left\{\frac{3}{2} \frac{S}{R_\infty(R)} + \frac{S}{R_\infty(I)}\right\} + \frac{3n_{\frac{1}{2},R}}{R_\infty(R)} + \frac{2n_{\frac{1}{2},I}}{R_\infty(I)}$$

$$= \frac{n}{S} + P\frac{n}{S} + Q,$$

met $P = \frac{3}{2} \frac{S}{R_\infty(R)} + \frac{S}{R_\infty(I)}$ en $Q = \frac{3n_{\frac{1}{2},R}}{R_\infty(R)} + \frac{2n_{\frac{1}{2},I}}{R_\infty(I)}$

We merken op dat de grootheden P en Q onafhankelijk zijn van n. Door het uitvoeren van 1 slag cyclische reductie is de rekentijd 2n/S, voor de oorspronkelijk recursie, teruggebracht tot n/S voor de overgebleven halve recursie en P(n/S) + Q voor de gevectori-

seerde andere helft. De waarde van P geeft hierin de winstfactor aan die behaald wordt door de helft van de recursie te vervangen door vectoriseerbare handelingen en Q geeft de opstarttijd weer voor dit vectorgedeelte, uitgedrukt in de $n_{\frac{1}{2}}$-waarden en R_{∞}-waarden van de respectieve onderdelen.

Het is duidelijk dat er slechts sprake van winst kan zijn als $P < 1$ en dan moet n bovendien nog zodanig zijn dat

$$P\frac{n}{S} + Q < \frac{n}{S}, \quad \text{ofwel} \quad \boxed{n > \frac{QS}{1-P}}$$

Op de resterende recursie van halve lengte kan natuurlijk ook weer cyclische reductie worden toegepast. De rekentijd voor deze halve recursie wordt dan

$$\frac{n}{2S} + P\frac{n}{2S} + Q$$

en de totale rekentijd voor het oplossen met 2 slagen cyclische recursie bedraagt nu

$$t_2 = \underbrace{\frac{n}{2S}}_{a} + \underbrace{P\frac{n}{2S} + Q}_{b} + \underbrace{P\frac{n}{S} + Q}_{c}$$

Deel a geeft de verwerkingstijd weer voor het resterende deel van de recursie (voor de x-waarden waarvan de index een viervoud is), b geeft de tijd weer voor de 2-de slag cyclische reductie en c representeert de tijd nodig voor de eerste slag. Men gaat nu weer eenvoudig na dat er (aangenomen dat $P < 1$) slechts sprake kan zijn van winst ten opzichte van 1 slag cyclische reductie, indien $n > 2QS/(1-P)$. Het is dus pas zinvol om 2 slagen uit te voeren indien n minstens twee keer zo groot is als de waarde waarvoor 1 slag nog zinvol was.

Op analoge wijze voortgaand, vindt men voor de rekentijd t_k na k slagen cyclische reductie

$$t_k = \frac{1}{2^{k-1}}\frac{n}{S} + P\frac{n}{S}\left(1 + \frac{1}{2} + \frac{1}{4} + \ldots.. + \frac{1}{2^{k-1}}\right) + kQ$$

en, mits $P < 1$, treedt er pas tijdwinst op ten opzichte van k-1 slagen cyclische reductie, als

$$n > \frac{2^{k-1}QS}{1-P}$$

Om elke volgende slag dus rendabel te maken, moet n steeds 2 keer zo groot zijn als de waarde waarvoor de voorgaande slag nog zin had. Voor zeer grote n en voor een voldoend grote k geldt, mits $k << n$, in relatief goede benadering

$$t_k = P\frac{n}{S}\left[\left(1+\frac{1}{2}+\frac{1}{4}+\ldots..+\frac{1}{2^{k-1}}\right)+\frac{1}{P2^{k-1}}+\frac{QSk}{Pn}\right]$$
$$\approx P\frac{2n}{S}$$

Voor de versnellingsfactor $F_k = t_0/t_k$ geldt blijkbaar $F_k \leq 1/P$. De waarde 1/P geeft een vaste bovengrens voor de te bereiken snelheidswinst via cyclische reductie.

Er schuilt bij deze analyse nog een flinke adder onder het gras. We hebben stilzwijgend aangenomen dat de prestatievermogenparameters R_∞ en $n_{\frac{1}{2}}$ voor elke reductiestap gelijk waren. Echter indien we het proces in de beschreven vorm toepassen, dan hebben we in de eerste reductieslag te maken met een stride 2, in de tweede slag met een stride 4 en in de k-de slag met een stride 2^k. Hoeveel geheugen banks een gegeven machine ook heeft, het is duidelijk dat we vroeg of laat in hevige mate te maken krijgen met memory bank conflicten en dat daardoor de waarden van R_∞ (R) en R_∞ (I) drastisch verlaagd kunnen worden. Men kan dit gelukkig enigszins voorkomen door de coëfficiënten van de gereduceerde recursies steeds aaneengesloten op te slaan. De berekening van deze coëfficienten bij de eerst reductieslag komt er dan bijvoorbeeld als volgt uit te zien:

$$\tilde{b}_j = b_{2j} + a_{2j} * b_{2j-1}$$
$$\tilde{a}_j = a_{2j} * a_{2j-1} \qquad j = 1, 2, 3, \ldots, \tfrac{n}{2}$$

en voor de volgende stappen analoog.

We laten nu voor een paar concrete gevallen zien hoe dit cyclische reductieproces uitpakt.

1. CRAY-1

Voor deze machine geldt grofweg dat de originele recursie wordt uitgevoerd met een snelheid van ongeveer 5 Mflops, dus S = 5. Voor het berekenen van de $\tilde{b}_j$'s en de $\tilde{a}_j$'s zijn vier vector load operaties vereist (de a_{2j}'s kunnen namelijk dubbel gebruikt worden), de flops kunnen gekoppeld worden en er zijn nog twee vector store operaties nodig voor het opbergen van de nieuwe coëfficiënten. Gemiddeld kunnen er dus drie flops per 6 klokcycli worden uitgevoerd en dientengevolge bedraagt R_∞ (R) ten naaste bij 3/6 * 80 = 40 Mflops (bij stride 2 treedt er op de CRAY-1 geen merkbare verandering in de vectorsnelheid op). Gezien onze eerder opgedane ervaringen, vermeld in voorgaande paragrafen, lijkt het niet onrealistisch om uit te gaan van een maximale snelheid die, ten gevolge van de stripmining, circa 10% lager uitvalt en dus R_∞ (R) ≈ 36 Mflops (natuurlijk is het een koud kunstje de werkelijke R_∞ (R) experimenteel te bepalen, de redenering hier is evenwel bruikbaar indien men niet over deze gegevens beschikt en toch alvast een ruwe analyse wenst uit te voeren). Ook voor R_∞ (I) schatten we op dezelfde wijze R_∞ (I) ≈ 36 Mflops. Voor P geldt dan:

$$P = \frac{3}{2} * \frac{5}{36} + \frac{5}{36} = \frac{25}{72} \approx \frac{1}{3}$$

Onder ideale omstandigheden, dat wil zeggen n groot en k ook groot en k << n, kunnen we dus met k slagen cyclische reductie de rekentijd met hooguit een factor 3 reduceren.

De snelheidswinst die met 1 slag cyclische reductie behaald kan worden bedraagt:

$$F_1 = \frac{t_0}{t_1} = \frac{2}{1+P+Q\frac{S}{n}} \leq \frac{2}{1+P} = 1.5$$

Verder geldt dat er met 1 slag pas werkelijk winst optreedt indien $n > QS/(1-)P$. Als we uitgaan van de realistische veronderstelling dat $n_{\frac{1}{2},R} \approx n_{\frac{1}{2},I}$, dan geldt $Q = (\frac{3}{36} + \frac{2}{36})\, n_{\frac{1}{2},R}$ en dus $n > \frac{25}{24}\, n_{\frac{1}{2},R}$.

Om winst te kunnen behalen met 1 slag cyclische reductie moet n dus een fractie groter zijn dan de vectorlengte waarvoor het reductiegedeelte zelf op de helft van de maximale snelheid ligt ($n_{\frac{1}{2},R} \approx 20$).

De verhouding tussen de vector en de scalaire snelheid (voor verschillende constructies weliswaar) is kennelijk van cruciale betekenis voor de effectiviteit van de cyclische reductie techniek. Omdat de CONVEX C-1 wat meer last heeft van stride problemen (bij het rekenen in de zogenaamde double precision = 64 bits arithmetiek) en dus die verhouding wat slechter zou kunnen uitpakken, bekijken we ook deze machine apart.

2. CONVEX C-1

Voor deze computer zouden we de analyse op dezelfde wijze kunnen uitvoeren als voor de CRAY-1, echter voor de verandering gaan we dit keer eens uit van de experimenteel bepaalde prestatievermogenparameters: S = .8 Mflops, $R_\infty(R) = R_\infty(I) = 2.4$ Mflops en $n_{\frac{1}{2},R} = n_{\frac{1}{2},I} \approx 8$. Dat levert $P = \frac{2}{3} * \frac{1}{3} + \frac{1}{3} = \frac{5}{6}$ en de zelfs in ideale gevallen te verwachten maximale winst is dus uiterst marginaal: een reductie van hooguit 16%. Eén van de redenen voor het beperkte succes van cyclische reductie is in dit geval dat de compiler zelf ook al wat scalaire optimalisatie heeft toegepast op de oorspronkelijke recursie, zodat er al een winst met een factor ≈ 1.6 ten opzichte van de 'echte' scalaire snelheid behaald wordt (ook voor kleine n-waarden). We moeten hieruit besluiten dat het niet de moeite loont, zelfs niet voor zeer hoge waarden van n, dit verder te optimaliseren met behulp van cyclische reductie.

3. IBM 3090/VF(1)

Ook voor deze computer gaan we weer uit van de experimenteel bepaalde snelheidsmodelwaarden: S = 6.5 Mflops en $R_\infty(R) = R_\infty(I) = 19$ Mflops (voor n ook weer niet te groot) en $n_{\frac{1}{2},R} = n_{\frac{1}{2},I} \approx 13$. Hieruit bepalen we voor P: $P = \frac{3}{2} * \frac{6.5}{19} + \frac{6.5}{19} \approx \frac{5}{6}$ en voor deze machine gelden dus soortgelijke overwegingen als voor de hiervoor besproken CONVEX C-1.

Voor machines met een grotere verhouding tussen de vector en scalaire snelheden, zoals de CRAY X-MP, FUJITSU VP-200 en de NEC SX-2, mag men wat meer effect

verwachten van de cyclische reductie techniek. Geleidelijk aan leveren alle fabrikanten ook compilers af waarin de oorspronkelijke recursie heel behoorlijk geoptimaliseerd wordt (vaak ook door achter de schermen een of andere vorm van cyclische reductie toe te passen) en dan loont het, zoals we al voor de CONVEX C-1 en IBM 3090/VF zagen, steeds minder de moeite om zelf aan de slag te gaan met deze cyclische reductie techniek.

Voor de CYBER 205 leidt de cyclische reductietechniek, in de hier beschreven vorm, ook niet rechtstreeks tot merkbare snelheidsverbetering, vanwege de stride $\neq 1$. In [51] wordt gedetailleerd aangegeven hoe scalaire verwerking van de recursie voor de CYBER 205 verder geoptimaliseerd kan worden met behulp van onder andere cyclische reductie.

Tenslotte zij nog vermeld, voor de fijnproevers, dat voor vectorregistermachines nog enige verdere verbetering van het proces gerealiseerd kan worden door twee slagen cyclische reductie samen te nemen. Hierdoor kunnen sommige tussenresultaten in de vectorregisters meermalen benut worden. Dit idee is voor de CRAY-1 nader uitgewerkt in [58], alwaar uitzicht wordt geboden op een te behalen snelheid van circa 13 Mflops met vier slagen cyclische reductie.

Het is uit deze beschouwingen duidelijk dat er met cyclische reductie voor dit soort recursies nog hooguit bescheiden winsten te realiseren zijn ten opzichte van de geoptimaliseerde scalaire algoritmen die doorgaans standaard voorhanden zijn. Voor andere problemen, bijvoorbeeld het oplossen van tridiagonale stelsels, kan dit evenwel weer heel anders uitpakken. Meer informatie over het cyclisch reduceren kan men vinden in [31].

Als laatste vectorisatietechniek bespreken we tenslotte de verdeel- en heerstechniek.

c. Verdeel- en heerstechniek

De fundamentele gedachte hier is dat men het gegeven probleem vervangt door een aantal onafhankelijke kleinere problemen, door op geschikte plaatsen koppelingen in het probleem achterwege te laten. Uiteraard geeft dit niet meteen, na oplossing van de deelproblemen, de gewenste oplossingen maar moeten achteraf de veronachtzaamde koppelingen eerst weer in rekening gebracht worden. De techniek is dan ook alleen met succes toepasbaar als deze correctieslag op relatief eenvoudige wijze kan plaatsvinden. Op conventionele computers heeft deze techniek, net zoals overigens recursive doubling en cyclische reductie, doorgaans weinig zin omdat het werk in de correctiefase niet goedgemaakt wordt door besparingen in de (onafhankelijke) oplosfasen. Er zijn evenwel problemen bekend waarbij toepassing van het verdeel-en-heersprincipe ook bij scalaire verwerking aanleiding geeft tot snellere algoritmen [9].

Wij passen de techniek weer toe op de inmiddels bekende recursie. Voor het gemak wordt verder aangenomen dat n te schrijven is als het produkt van twee gehele waarden m en k (die ongelijk zijn aan n zelf of aan 1). We gaan de lange recursie nu opbreken in m recursies, waarvan allen de lengte k hebben, door de elementen a_{k+1}, a_{2k+1}, a_{3k+1},... in de oorspronkelijke recursie door nullen te vervangen. Dat leidt tot de volgende m recursies:

$\tilde{x}_1 = b_1$	$\tilde{x}_{k+1} = b_{k+1}$		$\tilde{x}_{(m-1)k+1} = b_{(m-1)k+1}$
$\tilde{x}_2 = b_2 + a_2 * x_1$	$\tilde{x}_{k+2} = b_{k+2} + a_{k+2} * \tilde{x}_{k+1}$		
.		....	.
.			.
.			.
$\tilde{x}_k = b_k + a_k * x_{k-1}$	$\tilde{x}_{2k} = b_{2k} + a_{2k} * \tilde{x}_{2k-1}$		$\tilde{x}_{mk} = b_{mk} + a_{mk} * \tilde{x}_{mk-1}$

Wanneer de weggelaten coëfficiënten a_{k+1}, a_{2k+1},... zelf ongelijk aan nul zijn (en dat mogen we wel aannemen, anders hadden we de recursie zonder problemen kunnen opsplitsen), dan is wel duidelijk dat de oplossingen van elk deelprobleem, uitgezonderd het eerste, verschillen van de overeenkomstige segmenten van de gewenste x-waarden. Dat hebben we in het schema benadrukt door $\tilde{x}$ in plaats van x te schrijven. Om de gelijke vorm van de verschillende recursies te benadrukken hebben we ook de x-waarden voor de eerste groep van een ~ voorzien. Op dit moment zijn we evenwel nog niet zo geïnteresseerd in echte parallelle verwerkingsmogelijkheden (die komen later aan bod), maar nadere beschouwing leert ons dat het schema ook vectoroperaties mogelijk maakt. Hiervoor moeten we de opdrachten in het schema anders groeperen. We voeren eerst alle eerste stappen van de m recursies achter elkaar uit, daarna alle tweede stappen, enzovoort. In schema:

$\tilde{x}_j = b_j$, $j = 1, k+1, 2k+1, \ldots, (m-1)k+1$
$\tilde{x}_{j+1} = b_{j+1} + a_{j+1} * \tilde{x}_j$, $j = 1, k+1, 2k+1, \ldots, (m-1)k+1$
. . .
. . .
. . .
$\tilde{x}_{j+k-1} = b_{j+k-1} + a_{j+k-1} * \tilde{x}_{j+k-2}$, ,,

Elke regel in het schema representeert een vectoroperatie, immers x-en die rechts van het =-teken voorkomen zijn in de vorige operatie berekend. De schijnbare recursie kan dan ook,indien nodig, tot vectorisatie gedwongen worden door gebruik van bijvoorbeeld de eerder genoemde comment directieven. Elke vectoroperatie heeft de structuur

vector = vector + vector * vector

Dat ziet er bemoedigend uit en we gaan daarom snel verder met het onderzoek hoe de correctiefase plaats moet vinden. We bekijken daarvoor wat er gebeurt in de j-de recursiegroep en leiden een relatie af tussen de $\tilde{x}$-en en de werkelijke x-waarden. Voor deze groep geldt meteen dat:

$$x_{jk+1} = b_{jk+1} + a_{jk+1} * x_{jk} = \tilde{x}_{jk+1} + a_{jk+1} * x_{jk}$$

en

$$\begin{aligned} x_{jk+2} &= b_{jk+2} + a_{jk+2} * x_{jk+1} = \\ &= b_{jk+2} + a_{jk+2} * \tilde{x}_{jk+1} + a_{jk+2} * a_{jk+1} x_{jk} = \\ &= \tilde{x}_{jk+2} + a_{jk+2} * a_{jk+1} * x_{jk} \end{aligned}$$

Men gaat eenvoudig na dat algemeen geldt:

$$x_{jk+p} = \tilde{x}_{jk+p} + a_{jk+p} * a_{jk+p-1} * \ldots * a_{jk+1} * x_{jk}$$

Voor de correctie van de $\tilde{x}$-waarden hebben we dus slechts de waarde van x_{jk} (dat is de waarde van het laatste x-element van de voorgaande groep) nodig alsmede de geaccumuleerde produkten van de koppelingsparameters a. We slaan deze produkten op in een array c en de berekening van de elementen van c kan weer met vectoroperaties worden uitgevoerd door alle produkten met een gelijk aantal termen voor alle groepen achter elkaar uit te rekenen:

$c_j = a_j$ voor $j = 1, k + 1, 2k + 1, \ldots (m - 1)k + 1$
$c_{j+1} = a_{j+1} * c_j$ voor $j = 1, k + 1, \ldots\ldots$,,
. .
. .
. .
$c_{j+k-1} = a_{j+k-1} * c_{j+k-2}$,, ,,

Zowel de groepsgewijze oplosfase als de berekening van de 'correctie' coëfficiënten vindt plaats met k vectoroperaties werkend op vectoren ter lengte m (echter, bij 'normale' opslag van de elementen achter elkaar, ook met stride k).

De correctie van de elementen x kan nu op twee verschillende manieren worden uitgevoerd.

I.
Groep na groep wordt gecorrigeerd. De x-waarden die in de eerste groep zijn opgeleverd zijn al meteen de correcte waarden en behoeven dus geen correctie. Voor groep 2 geldt:

$$\tilde{x}_{k+1} = x_{k+1} + c_{k+1} * x_k$$
$$\tilde{x}_{k+2} = x_{k+2} + c_{k+2} * x_k$$
$$\vdots$$
$$\tilde{x}_{2k} = x_{2k} + c_{2k} * x_k$$

Als we nu meteen de oude $\tilde{x}$-waarden door de gewenste x-waarden overschrijven, dan staat hierboven niets anders dan de reeds uitvoerig besproken vector update of SAXPY (immers x_k is een constante in de boven geschetste rij opdrachten). Op analoge wijze kunnen nu de waarden van de derde groep gecorrigeerd worden en zo wordt groep na groep opgerold met een eenvoudige SAXPY.

Deze aanpak leidt tot in totaal m-1 vectoroperaties van elk lengte k (en stride 1). Bedenk wel dat in de eerste fase vectoroperaties optraden met lengte m. Als nu m en k zeer ongelijk van grootte zijn en één van beide is klein, dan is duidelijk dat één van beide fasen in het proces geen hoge (vector)snelheden toelaat. Dit bezwaar wordt (gedeeltelijk) ondervangen door aanpak II.

II.
Uitgaande van de (bekende) waarde van x_k, na afloop van het uitvoeren van de m recursies, corrigeren we nu eerst de laatste $\tilde{x}$-waarden van elke groep:

$$\begin{aligned} x_{2k} &= \tilde{x}_{2k} + c_{2k} * x_k \\ x_{3k} &= \tilde{x}_{3k} + c_{3k} * x_{2k} \\ &\vdots \\ x_{mk} &= \tilde{x}_{mk} + c_{mk} * x_{(m-1)k} \end{aligned}$$

Dit is weliswaar een (echte) recursie, echter zij is van lengte m en dus k keer zo klein als de oorspronkelijke recursie. Als k nu maar niet al te klein is, dan hopen we vooralsnog dat de schade beperkt blijft. Nadat via deze recursie alle x_{jk} bekend zijn kunnen voor alle groepen de $\tilde{x}$-waarden simultaan (dat wil zeggen onafhankelijk van elkaar) gecorrigeerd worden. Dit wordt nu weer uitgevoerd door eerst alle eerste waarden van alle groepen te corrigeren, daarna alle tweede waarden, enzovoort:

$$\begin{aligned} x_{jk+1} &= \tilde{x}_{jk+1} + c_{jk+1} * x_{jk} && , j = 1, \ldots , m-1 \\ x_{jk+2} &= \tilde{x}_{jk+2} + c_{jk+2} * x_{jk} && , j = 1, \ldots , m-1 \\ &\vdots && \vdots \\ x_{jk+k-1} &= \tilde{x}_{jk+k-1} + c_{jk+k-1} * x_{jk} && , j = 1, \ldots , m-1 \end{aligned}$$

Dit zijn k-1 vectoroperaties op vectoren van lengte m-1 (en stride k), dus nagenoeg dezelfde situatie als in de groepsgewijze oplosfase.

We kunnen nu, na deze chaotisch ogende rekentaferelen, de balans gaan opmaken. De oorspronkelijke recursie die ongeveer 2n flops omvatte, is door onze verdeel-en- heerstechniek vervangen door de volgende operaties:

C1. De verdeel- en oplosfase
- de berekening van de $\tilde{x}$-en:
  k vectoroperaties met elk vectorlengte m: $\approx$ 2mk flops
- de berekening van de correctiecoëfficiënten c:
  k vectoroperaties met elk vectorlengte m: $\approx$ mk flops

C2. De correctiefase
Aanpak I : m-1 vectoroperaties ter lengte k: $\approx$ 2mk flops
Aanpak II : – een recursie ter lengte m: $\approx$ 2m flops
– k-1 vectoroperaties met vectorlengte m-1: $\approx$ 2mk flops
Ook als men het aantal flops wat nauwkeuriger narekent, blijkt het verdeel-en-heersalgoritme nagenoeg 5n flops te kosten (in plaats van de ongeveer 2n flops voor de oorspronkelijke recursie). Het is duidelijk dat voor dit probleem het verdeel-en-heersalgo-

ritme geen goed alternatief biedt op conventionele scalaire computers. Op vectorcomputers (en parallelle computers) ligt dat echter anders.

We zullen nu implementatie en rekensnelheid van dit algoritme wat nader bekijken. Bij de hier besproken toepassing wordt aangenomen dat de invoerwaarden van a_j en b_j overschreven mogen worden als dat wenselijk mocht zijn. In het geval dat de a_j's bewaard moeten blijven om er volgende recursies met gewijzigde b_j-waarden mee door te rekenen, kan er op het algoritme wat bespaard worden door de correctiecoëfficiënten c eenmalig uit te rekenen en deze naast de a_j's op te slaan. Wij zullen achtereenvolgens de b_j's eerst overschrijven door de waarden van de x_j's en daarna door die van de gewenste x_j's. De waarden van de a_j's zullen we overschrijven door die van de c_j's.

Om de groepsgewijze verdeling van de oorspronkelijke recursie in m kleinere recursies beter zichtbaar te maken, slaan we de a_j's en b_j's op in tweedimensionale arrays A(K,M) en B(K,M). De j-de kolom van bijvoorbeeld A geeft dan precies de coëfficiënten aan die horen bij de j-de recursiegroep. Bedenk hierbij dat in FORTRAN de kolommen achter elkaar in het geheugen worden opgeslagen, zodat alle kolommen achter elkaar juist overeenkomen met het oorspronkelijke ééndimensionale array dat de originele recursie weergeeft.

De verdeel- en oplosfase komt er in FORTRAN als volgt uit te zien:

```
      DO 20 I = 2,K
        DO 10 J = 1,M
          B(I,J) = B(I,J) + A(I,J) * B(I-1,J)
          A(I,J) = A(I,J) * A(I-1,J)
10      CONTINUE
20    CONTINUE
```

(De eerste rij van A en B hoeft niet veranderd te worden.)
De correctie-elementen A(I,1) hoeven niet berekend te worden. Om dubbel gebruik te kunnen maken van de A(I,J) hebben we deze berekening samengenomen met die van B en daarbij geen uitzondering gemaakt voor J = 1. De correctiefase volgens aanpak I wordt dan:

```
      DO 40 J = 2,M
        DO 30 I = 1,K
        B(I,J) = B(I,J) + C(I,J) * B(K,J-1)
30      CONTINUE
40    CONTINUE
```

De DO 10-loop werkt op vectoren met stride K, de DO 30-loop daarentegen werkt op vectoren met stride 1. Indien we de arrays A en B rijgewijs zouden hebben opgeborgen, dan zou de situatie precies omgekeerd zijn met betrekking tot de strides, dus daar schieten we niet zo veel mee op.

We kunnen dit algoritme in principe weer net zo analyseren als de cyclische reductietechniek. We zullen hier echter volstaan met onszelf een indruk te verschaffen van de

mogelijkheden aan de hand van een globale beschouwing voor de CRAY-1. We nemen aan dat K geen veelvoud is van 4, zodat geheugenbankconflicten buiten beschouwing kunnen blijven.

In de DO 10-loop moeten de volgende vectoren worden opgehaald: de I-de rij van zowel A als B, alsmede de (I-1)-ste rij van beide arrays (de I-de rij van A kan dubbel worden gebruikt). Vervolgens moeten de I-de rijen van A en B, na bijwerking, weer uit de registers naar het geheugen worden teruggeschreven. Aangezien de * en + operaties weer parallel aan de vectorlaadoperaties kunnen worden uitgevoerd, mag men voor voldoende grote M een gemiddelde snelheid van ongeveer 3 flops per 6 klokcycli verwachten. Evenzo mag men, als K ook maar weer voldoende groot is, voor de DO 30-loop een verwerkingssnelheid verwachten in de buurt van 2 flops per 3 klokcycli (zie de SAXPY discussie voor de CRAY-1 in paragraaf 7.1). Totaal verwerkt het algoritme dus ongeveer 5 flops per 9 klokcycli en dus zou de maximale rekensnelheid ongeveer 5/9 * 80 ≈ 44.4 Mflops kunnen bedragen.

Omdat het verdeel- en heersalgoritme ongeveer 5n flops vergt, bedraagt de rekentijd, uitgedrukt in µsec., ongeveer 5n/44.4, tegen een CPU-tijd van ruwweg 2n/5 µsec. voor de originele recursie (rekensnelheid circa 5 Mflops). De tijdwinst bedraagt dus maximaal een factor $\frac{2n}{5} / \frac{5n}{44} \approx 3.6$.

Voor K = 305 en M = 305 (dus n = 305^2) is dit geprobeerd en toen bleek de werkelijke winst ongeveer 3 te bedragen. Uiteraard hangt dit sterk af van de waarden van K en M en laat voor lagere waarden hiervan de opstarttijd zich nadrukkelijker gelden.

Programmerend in Assembleertaal zou het nog mogelijk moeten zijn de I-de rij van A en B in de vectorregisters vast te houden zodat deze waarden in de (I+1)-ste doorgang van de DO 10-loop weer gebruikt kunnen worden. Dit spaart twee vectorlaadoperaties uit en het algoritme zou er dus ongeveer 9/7 keer sneller op kunnen worden.

Bij vectorcomputers die betere toegangsmogelijkheden tot het geheugen hebben, mogen we uiteraard meer flops per klokcyclus verwachten voor dit algoritme en als bovendien de klokcyclus zelf ook nog weer korter is, dan zijn de te verwachten winstfactoren navenant hoger. In [59] is dit vrij uitvoerig geanalyseerd en daar worden winstfactoren van ongeveer 8 voor de CRAY X-MP en ongeveer 16 voor de FUJITSU VP-200 ten opzichte van de scalaire code voor de recursie gerapporteerd.

Aanpak II werkt beter voor vectorcomputers die problemen ondervinden van strides ≠ 1, zoals de FUJITSU VP-200, de CONVEX C-1 en natuurlijk de CYBER 205. Voor deze laatste machine zullen we aanpak II eens wat uitvoeriger analyseren. Om stride problemen te vermijden zorgen we ervoor dat de arrays A en B rijgewijs worden opgeborgen. Dit doen we door de oorspronkelijke arrays te spiegelen zodat daarna de j-de kolom van bijvoorbeeld A de coëfficiënten herbergt horend bij de j-de vergelijking van elke groep (het had ook gekund door gebruik te maken van de speciale CONTROL DATA declaratie 'ROWWISE'). De verdeel- en groepsgewijze oplosfase ziet er dan als volgt uit:

```
      DO 30 I = 2,K
        D0 10 J = 1,M
          B(J,I) = B(J,I) + A(J,I) * B(J,I-1)
10      CONTINUE
        DO 20 J = 2,M
```

```
          A(J,I) = A(J,I) * A(J,I-1)
20      CONTINUE
30    CONTINUE
```

De correctiefase volgens aanpak II wordt nu:

```
      DO 40 J = 2,M
        B(J,K) = B(J,K) + A(J,K) * B(J-1,K)
40    CONTINUE
      DO 60 I = 1,K-1
        DO 50 J = 2,M
          B(J,I) = B(J,I) + A(J,I) * B(J-1,K)
50      CONTINUE
60    CONTINUE
```

Veilige vectorisatie van de DO 10-, DO 20- en DO 50-loops kan worden afgedwongen via speciale CONTROL DATA FORTRAN Vector Syntax. De DO 10-loop wordt daarmee vervangen door de vectoropdracht

B(1,I;M) = B(1,I;M) + A(1,I;M) * B(1,I-1;M)

Met B(1,I;M) wordt een vector van M elementen aangeduid, die begint met het element B(1,I). De overige loops dienen op overeenkomstige wijze te worden omgeprogrammeerd.

De rekensnelheid van de CYBER 205 voor dit algoritme analyseren we nu eens door een gooi te doen naar het aantal benodigde klokcycli. In principe kan de snelheid ook geanalyseerd worden met behulp van parameters R_∞ en $n_{\frac{1}{2}}$ of door een asymptotische beschouwing zoals we hiervoor, met name voor de CRAY-1, hebben gedemonstreerd.

Een enkelvoudige vectoroperatie op een 2-pipe CYBER 205 neemt 51 klokcycli opstarttijd in beslag, gevolgd door N/2 klokcycli voor de vectorlengte N (neem maar aan dat N even is, anders komt er nog een klokccyclus bij). De DO 10-loop valt uiteen in twee simpele vectoroperaties, eerst moeten namelijk A(J,I) en B(J,I-1) met elkaar vermenigvuldigd worden en daarna pas moet het weggeschreven resultaat weer worden opgehaald om bij B(J,I) te worden opgeteld. Omdat de DO 10-loop zelf ook weer K-1 keer moet worden uitgevoerd vergt dit onderdeel in totaal 2(K-1)M/2 + 2(K-1)51 klokcycli.

De ook (K-1) keren uit te voeren DO 20-loop vergt alles bij elkaar (K-1)(M-1)/2 + (K-1)51 klokcycli, zodat het aantal klokcycli voor de gehele DO 30-loop ongeveer (K-1)(M-1) + (K-1)M + 3(K-1)51 klokcycli gaat bedragen (afgezien van extra kosten voor de loop zelf en afgezien van mogelijke overlapping van de opstarttijden). De recursie in de DO 40–loop wordt uitgevoerd in 2(M-1)S klokcycli, waarbij S het aantal klokcycli per flop voor deze niet vectoriseerbare loop aangeeft.

Voor de totale DO 60-loop gaat men gemakkelijk na, op dezelfde wijze als voor de DO 30-loop, dat het totale aantal klokcycli ongeveer (K-1)(M-1) + 2(K-1)51 moet bedragen. Het totale aantal klokcycli voor het verdeel- en heersalgoritme op de CYBER 205 geven we aan met N_c en hiervoor geldt dus:

$$N_c = 5n/2 + (2S-5/2)M + 253.5\ n/M - 2S - 253.5, \quad (\text{met } n = MK)$$

Omdat de lengte van een klokcyclus 20 ns bedraagt, kunnen we de totale rekentijd dus eenvoudig berekenen. Door M relatief groter te kiezen, wordt de tweede term in N_c relatief groter en zakt de derde term juist. Differentiëren van N_c naar M levert de waarde van M waarvoor de verwerkingstijd minimaal is:

$$M = \sqrt{\frac{253.5}{2S-2.5}} \cdot \sqrt{n}$$

Voor S hebben we experimenteel waarden in de buurt van 6.5 klokcycli vastgesteld, zodat de ideale waarde van M ongeveer 4.9 $\sqrt{n}$ zou moeten bedragen.

Het algoritme werd daadwerkelijk uitgevoerd voor n = 60480, met M = 945 en K = n/945 = 64 (de ideale waarde van M zou ongeveer 1208 moeten zijn, echter dat is niet deelbaar op n).

De theoretisch te verwachten CPU-tijd bedraagt dan $N_c * 20_{10}^{-9}$ sec. $\approx 3.54_{10}^{-3}$ sec. De werkelijk gemeten CPU-tijd bedroeg 3.63_{10}^{-3} sec. en dat komt dus vrij aardig overeen met de theoretische waarde.

Tenslotte, om een indruk te geven van de behaalde tijdwinst, de rekentijd voor de oorspronkelijke recursie ter lengte n bedroeg 3.14_{10}^{-2} sec. en het verdeel-en-heersalgoritme leidt dus in ongeveer 8.4 keer kortere tijd tot het gewenste resultaat.

De geïnteresseerde lezer kan in de literatuur meer gegevens vinden over het verdeel-en-heersalgoritme, toegesneden op lineaire stelstels vergelijkingen. Chen, Kuck en Sameh [4] beschreven de methode voor algemene benedendriehoeksmatrices met een bandstructuur. De meeste verwijzingen treft men evenwel aan naar een later verschenen artikel van Wang, die de methode toepaste op tridiagonale stelsels vergelijkingen [60]. Duff, Erisman en Reid [23] beschreven voor dezelfde soort vergelijkingen een met onze aanpak II vergelijkbare versie. In [59] is de methode uitvoerig beschreven voor toepassing op lineaire recursies (of bidiagonale stelsels).

Een belangrijk aspect dat hier tot nu toe onbesproken is gebleven betreft de numerieke stabiliteit van de beschreven algoritmen. We hebben met het oog op efficiency verschillende technieken beschreven, echter het is niet zonder meer duidelijk of deze algoritmen resultaten afleveren met een vergelijkbare precisie als de oorspronkelijke recursie. Het zou best zo kunnen zijn dat de vectoriseerbare alternatieven veel (on)gevoeliger zijn ten aanzien van afrondfouten. Vooral het verdeel- en heersalgoritme heeft de naam in het algemeen slechtere resultaten af te leveren. In [59] is evenwel aangetoond dat voor een veel voorkomende klasse van lineaire recursies (namelijk die waarvoor geldt $|a_1| \leq 1$), het verdeel- en heersalgoritme (ook wel: partition-methode) resultaten van nagenoeg gelijke kwaliteit aflevert als de oorspronkelijke recursie.

NEGEN

HERSTRUCTURERING VAN ALGORITMEN EN DATA-ORGANISATIE

In voorgaande hoofdstukken hebben we min of meer uitvoerig kennis gemaakt met de mogelijkheden die door vectorrekenen geboden worden. Om de snelheid van dit type architectuur optimaal te benutten, moet men er in feite voor zorgen dat een algoritme zoveel mogelijk uitgedrukt wordt in termen van regelmatige en lange vectoroperaties of, liever nog, in termen van matrixvectoroperaties. Ook bij echte parallelle verwerking wenst men de basiscomponenten van een algoritme niet al te klein te kiezen, om allerlei overhead te beperken, en dan komt men ook al gauw op het niveau van matrixvectoroperaties terecht. Het moge duidelijk zijn dat het in alle gevallen aantrekkelijk is om een gegeven algoritme uit te schrijven in zo groot mogelijke deeloperaties. Immers naar mate de deeloperaties groter zijn, bieden zij meer mogelijkheden tot optimalisatie voor een gegeven specifieke computerarchitectuur.

De oorspronkelijke BLAS-collectie [43] bestaat uit relatief eenvoudige vectoroperaties, zoals de vectorupdate (SAXPY, zie hoofdstuk 7) en het inprodukt (SDOT, zie hoofdstuk 7). Mede onder invloed van het toenemend gebruik van vectorsupercomputers heeft men zich gerealiseerd dat deze kernen in feite te kleinschalig zijn en heeft men de inmiddels vrij algemeen geaccepteerde Extended BLAS gedefiniëerd [13], ook wel Level 2-BLAS genoemd. Deze Extended BLAS verzameling is samengesteld uit matrix-vector operaties. Voor een aantal supercomputers zijn deze grotere basismodulen standaard in geoptimaliseerde vorm beschikbaar voor gebruik in FORTRAN-programmatuur. Door gecompliceerde algoritmen zoveel mogelijk uit te drukken in deze modulen is men dan verzekerd van een vrij hoge mate van efficiency zonder dat men zichzelf met daadwerkelijke vectorisatieaanpassingen in het programma hoeft bezig te houden. Ook bespaart men zich de moeite om bij overgang naar een ander type vectorcomputer de hele zaak opnieuw te vectoriseren of anderszins te tunen. In het ergste geval kan men zich beperken tot het zelf optimaliseren van de benodigde Extended BLAS-modulen, mocht de fabrikant daar niet voor gezorgd hebben.

Doordat supercomputers, hetzij vector, hetzij parallel van opzet, of beide, het mogelijk maken om zeer grote problemen door te rekenen, werd het al ras duidelijk dat ook de Level 2 BLAS nog te beperkt van opzet zijn. Eén van de redenen is dat het optimale gebruik van data wat gehinderd wordt indien men de deeloperaties te kleinschalig kiest. We zullen daar een uitvoerig voorbeeld van laten zien in hoofdstuk 10. Bij grotere problemen dient men zorgvuldig met de beschikbare geheugens (en de transporten daartussen) om te gaan en stuit men al gauw op de verschillen tussen snel en traag geheugen.

Daarom heeft men een poging ondernomen om een beperkt aantal basisoperaties te definiëren die dusdanig groot zijn dat zij voldoende mogelijkheden bieden met betrekking tot zowel efficiënte verwerking op vector- en parallelle computers als ook tot efficiënt geheugenbeheer. Anderzijds moesten deze basisoperaties nog voldoende elementair zijn, opdat veel voorkomende algebraïsche bewerkingen erin uitgedrukt kunnen worden. Dit heeft vooralsnog geleid tot een voorstel voor de zogenaamde Level 3 BLAS [15]. De voorgestelde collectie omvat momenteel vier hoofdgroepen van basisroutines:

a. matrix-matrixvermenigvuldiging;
b. bijwerken van een matrix met een matrix van lagere rang;
c. vermenigvuldiging van een matrix met een driehoeksmatrix;
d. vermenigvuldiging van een matrix met de inverse van een driehoeksmatrix.

We zullen in dit hoofdstuk een voorbeeld geven van de herstructurering van een algoritme in termen van steeds grotere deeloperaties. Hiervoor kiezen we het oplossen van een lineair stelsel vergelijkingen $Ax = b$ en meer in het bijzonder beperken we ons tot de constructie van de daarvoor benodigde matrixontbinding van A. Om de beschrijving eenvoudig te houden en allerlei stabiliteitsproblemen te ontwijken, nemen we aan dat A symmetrisch is en positief definiet. De meest tijdrovende stap bij directe oplossing (in tegenstelling tot iteratieve oplossing) is de constructie van de zogenaamde Choleski-ontbinding van A: $A = LL^T$, waarbij L een benedendriehoeksmatrix is.

Een standaardmanier om L te berekenen is de volgende. Men schrijft de matrix A formeel als het produkt van L en L^T:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdot \\ a_{31} & & & \cdot \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ a_{n1} & \cdot & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \varnothing \\ l_{31} & l_{32} & l_{33} & \\ \cdot & & \cdot & \\ \cdot & & & \cdot \\ l_{n1} & \cdot & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} \cdot & \cdot & \cdots & \cdot \\ & \cdot & & \cdot \\ & & \cdot & \cdot \\ \varnothing & & \cdot & \cdot \\ & & & \cdot \end{bmatrix}$$

A L L^T

De kolommen van L^T zijn precies gelijk aan de rijen van L. Stel nu dat de eerste j-1 rijen van L inmiddels bekend zijn. Door het inprodukt te nemen van de j-de (nog onbekende) rij van L met achtereenvolgens de eerste tot en met de (j-1)-ste kolom van L^T, en deze inprodukten gelijk te stellen aan elementen van A (in de j-de rij) ontstaan relaties waaruit de waarden van l_{jk}, $k = 1, \ldots, j - 1$, berekend kunnen worden:

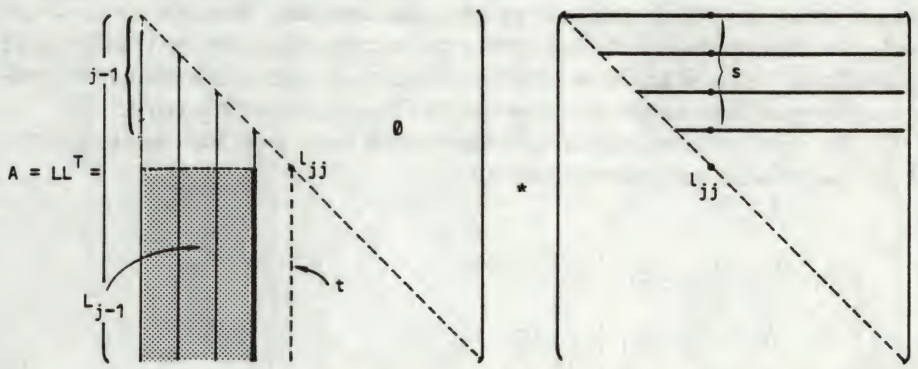
$$l_{jk} = (a_{jk} - \sum_{i=1}^{k-1} l_{ji}\, l_{ki}) / l_{kk}$$

En uit het inprodukt van de j-de rij met zichzelf ontstaat:

$$l^2_{jj} = (a_{jj} - \sum_{i=1}^{j-1} l^2_{ji})$$

We zien dat het rekenwerk bij de Choleski decompositie grotendeels bestaat uit het berekenen van inprodukten (BLAS:SDOT). Bij vectorverwerking zullen we er bij voorkeur voor moeten zorgen dat de elementen van L *rijgewijs* achter elkaar staan opgeborgen om stride problemen te voorkomen. Men kan dat eventueel ook bewerkstelligen door in plaats van de rijen van L de kolommen van L^T te berekenen, indien men de voorkeur geeft aan opslag der elementen in een vierkant, dat wil zeggen dubbel geïndiceerd, array.

Een andere manier van uitvoering wordt verkregen door achtereenvolgens de kolommen van L te berekenen (en dus de rijen van L^T). We lichten dit toe aan de hand van de onderstaande figuur:



We veronderstellen weer dat inmiddels de eerste j-1 kolommen van L bekend zijn en dat we nu voor de taak staan de j-de kolom te berekenen. L_{j-1} geeft het deel van L aan dat bestaat uit deze eerste j-1 kolommen minus hun eerste j-1 rijen. De vector t is gelijk aan het niet-nul gedeelte van de j-de (onbekende) kolom: $t = (l_{jj}, l_{j+1,j},..., l_{n,j})^T$ en met s geven we de vector aan die bestaat uit de eerste j-1 elementen van de j-de kolom van L^T. Tenslotte noteren we met a het met t overeenkomstige gedeelte van de matrix A, dus $a = (a_{jj}, a_{j+1,j},..., a_{n,j})^T$. Door nu het inprodukt te nemen van het gedeelte $(L_{j-1};t)$ van L met de vector $(s;l_{jj})$ uit L^T en dat gelijk te stellen aan de betreffende elementen van A, krijgen we de volgende relatie:

$$L_{j-1}s + l_{jj}t = a$$

ofwel

$$l_{jj}t = a - L_{j-1}s.$$

Uit deze laatste relatie zijn vrij eenvoudig de elementen van de vector t te berekenen. We zien dat, mits j niet al te klein is, het grootste deel van het rekenwerk nu gaat zitten in de berekening van L_{j-1}s en dat is een matrix-vectorbewerking. We zijn nu dus op het niveau van de Level 2-BLAS geraakt. Men kan zelfs nog de aftrekking van de vector a in zo'n Level 2-BLAS meenemen, dat wil zeggen dat de hele operatie a-L_{j-1}s gezien kan worden als een Level 2-BLAS kern (een soort gegeneraliseerde SAXPY).

Voor een efficiënte verwerking via deze Level 2-BLAS modulen is het, in ieder geval voor dit voorbeeld, nu wel gewenst om van een rijgewijze opslag van L over te stappen op een kolomsgewijze opslag. Dat houdt in dat, om effectief van de Level 2-BLAS gebruik te kunnen maken, het kan voorkomen dat men de data op een andere wijze dan voorheen moet gaan rangschikken. Dit kan in de praktijk een vrij vervelende en ingrijpende operatie op een bestaand programma inhouden. Immers het oplossen van een stelsel is vaak maar een (klein) deelprobleem binnen een veel grotere modelberekening en dat zou er toe kunnen leiden dat men de hele organisatie van een ingewikkeld programma opnieuw moet bezien.

In deze zin is het hier gegeven voorbeeld illustratief voor ingewikkelder praktijksituaties. Het vectoriseren danwel parallel maken van een gegeven algoritme is meestal niet het grootste werk, de meeste inspanning gaat doorgaans zitten in een vrijwel volledige herstructurering van de data (en dus van het programma) om een redelijke mate van efficiency te bereiken. Dit wordt nog duidelijker naar voren gebracht door de nu volgende reorganisatie van het algoritme voor de Choleski-ontbinding van A.

We gaan de betrokken matrices onderverdelen in vrij grote submatrices, bijvoorbeeld (aannemend dat n een viervoud is):

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix} = \begin{bmatrix} L_{1,1} & & & \\ L_{2,1} & L_{2,2} & & \\ L_{3,1} & L_{3,2} & L_{3,3} & \\ L_{4,1} & L_{4,2} & L_{4,3} & L_{4,4} \end{bmatrix} \times L^T$$

We kunnen de submatrices van L weer in (super)kolomsgewijze volgorde berekenen:

$$A_{1,1} = L_{1,1}\ L_{1,1}^T$$

Hieruit volgt $L_{1,1}$ volgens de hiervoor geschetste kolomsgewijze aanpak (via Level 2-BLAS).

Verder geldt:
$$A_{2,1} = L_{2,1}\ L_{1,1}^T$$
$$A_{3,1} = L_{3,1}\ L_{1,1}^T$$
$$A_{4,1} = L_{4,1}\ L_{1,1}^T$$

ofwel:
$$\begin{bmatrix} L_{2,1} \\ L_{3,1} \\ L_{4,1} \end{bmatrix} = \begin{bmatrix} A_{2,1} \\ A_{3,1} \\ A_{4,1} \end{bmatrix} L_{1,1}^{-T}$$

Deze laatste en meest tijdrovende stap (als het aantal blokken niet te klein is) is van het type

matrix = matrix * inverse van driehoeksmatrix

en behoort daarmee tot de vierde hoofdgroep van de Level 3-BLAS.

Na deze stap gaan we op voor de hand liggende wijze verder met de berekening van de tweede matrixkolom van L:

$$L_{2,1} L_{2,1}^T + L_{2,2} L_{2,2}^T = A_{2,2}$$

en dus:

$$L_{2,2} L_{2,2}^T = A_{2,2} - L_{2,1} L_{2,1}^T$$

Aan de rechterzijde vinden we een operatie op matrices en voor de berekening van $L_{2,2}$ zelf kunnen we weer met Level 2-BLAS te werk gaan. De rest van het algoritme verloopt verder op soortgelijke wijze.

We merken voor de goede orde nog op dat door deze drie verschillende manieren van algoritme-organisatie het aantal floating point operaties gelijk blijft. Het enige wat in feite verandert is de benodigde datastructuur. We zullen in het volgende hoofdstuk laten zien dat het van groot belang is een goede dataorganisatie op te zetten en dat de in de laatste aanpak gedemonstreerde bloksgewijze datastructuur niet alleen voldoende aanknopingspunten biedt voor vector en parallelle verwerking (denk aan de verdeling van de blokken over bijvoorbeeld locale geheugens), maar ook ruimte laat om de hoeveelheid transport tussen snel en traag geheugen (bijvoorbeeld schijven-centraal geheugen of centraal geheugen-cache of centraal geheugen-vectorregisters, of combinaties daarvan) te minimaliseren.

TIEN

EFFICIËNT GEHEUGENGEBRUIK

Het materiaal in dit hoofdstuk is vrijwel geheel gebaseerd op publikaties van Winter [63] en McKellar en Coffman [44].

Vrijwel alle moderne computers maken op een of andere wijze gebruik van verschillende soorten geheugen waarin data (tijdelijk) worden opgeslagen. Het gebruik van vectorregisters is reeds uitgebreid aan de orde geweest en ook hebben we, met name bij de matrixvectorvermenigvuldiging (paragraaf 7.3) gezien hoe deze vectorregisters beter benut konden worden zodat de hoeveelheid datatransport, en daarmee ook de CPU-tijd, gereduceerd kon worden. Een tussenvorm van geheugen is het zogenaamde cachegeheugen, een wat kleiner geheugen waarin data tijdelijk kunnen worden opgeslagen om bijvoorbeeld een regelmatigere en snellere toevoer tot de vectorregisters te helpen bewerkstelligen.

Voor de grote hoeveelheden data die vaak vereist zijn bij grootschalig rekenwerk zijn de meeste centrale geheugens bij lange na niet toereikend en moet er gebruik gemaakt worden van andere opslagmogelijkheden zoals het (goedkopere) Solid State Disc geheugen bij CRAY en een soortgelijk geheugen bij de NEC SX-2, of, wat meestal het geval is, snelle schijfgeheugens (echter traag vergeleken met het centrale geheugen). In feite spelen deze opslagproblemen natuurlijk ook al een rol op veel kleinere systemen, zoals bijvoorbeeld PC's waarbij grote hoeveelheden data via floppy disks of magneetbandjes gemanipuleerd moeten worden. De problematiek dateert dan ook niet van vandaag of gisteren, de publicatie van McKellar en Coffman verscheen bijvoorbeeld in 1969 en ook lang daarvoor was er al veel gepubliceerd met betrekking tot het uitvoeren van rekenwerk op data die uit een langzamer (achtergrond)geheugen gehaald moesten worden. Zie bijvoorbeeld ook het beroemde standaardwerk van Wilkinson [62, bladzijde 294].

Het gebruik van achtergrondgeheugen wordt op de meeste moderne computersystemen automatisch geregeld via het zogenaamde virtual memory concept. Men

declareert in het programma net zoveel werkruimte als men denkt nodig te hebben zonder daarbij rekening te houden met de fysieke capaciteit van het centrale geheugen en het bedrijfssysteem van de computer zorgt er zelf voor dat die gedeelten van de data die op zeker moment nodig zijn voor de berekeningen ook daadwerkelijk naar het centraal geheugen getransporteerd worden (en daar minstens net zo lang blijven als nodig is).

Alhoewel een en ander automatisch geregeld wordt is het toch zaak om hier bij omvangrijke databestanden terdege bewust te zijn van wat er zich achter de schermen afspeelt, zoals we nu zullen laten zien aan de hand van een voorbeeld. Dit voorbeeld is weliswaar opgehangen aan het virtual memory systeem van een specifieke computer, soortgelijke problemen spelen evenwel ook bij andere computers; niet in het minst wanneer men zelf het geheugenmanagement (bijvoorbeeld van en naar disk of magneetband; de zogenaamde I/O) organiseert. We zullen eerst iets meer vertellen over het virtual memory concept.

Bij virtual memory zijn de data opgeslagen op zogenaamde pages (pagina's dus). Machineïnstructies verwijzen naar virtuele adressen van operanden en zo'n virtueel adres bestaat uit het paginanummer, van de pagina waarop de bedoelde operand zich bevindt, alsmede het adres van de operand op de pagina zelf. Het systeem houdt bij welke pagina's er actueel in het centraal geheugen aanwezig zijn en indien er gebruik gemaakt moet worden van data die zich op een niet in het centraal geheugen aanwezige pagina bevinden, dan wordt de betreffende pagina in zijn geheel binnengehaald en overschrijft dan normaliter de ruimte in het centraal geheugen die voordien werd ingenomen door een eerder benodigde pagina. Gewoonlijk wordt dan de pagina die het langst buiten gebruik was in het centrale geheugen overschreven. Voor elk programma wordt een zogenaamde paginatabel bijgehouden die vermeldt waar een pagina zich op ieder moment bevindt (centraal geheugen of achtergrondgeheugen).

Indien de pagina in het centrale geheugen ligt dan wordt bovendien bijgehouden of er sedert binnenkomst in dat geheugen data op de pagina veranderd zijn (immers als er niets op de pagina veranderd is, dan hoeft deze niet expliciet naar achtergrondgeheugen teruggeschreven te worden, maar kan volstaan worden met de verwijzing naar de oorspronkelijke plaats van de pagina in het achtergrondgeheugen) en er wordt bijgehouden welke pagina het laatst gebruikt werd, welke het één na laatst, enzovoort.

We houden nu verder het voorbeeld van Winter [63] aan, beschreven voor de CYBER 205. Bij de CYBER 205 kan de dataopslag op zogenaamde 'large pages' georganiseerd worden, elke large page omvat 65 536 woorden van 64 bits. Het binnenhalen van een nieuwe large page in het geheugen wordt een '(large) page fault' genoemd en kost op de CYBER 205 fysiek ongeveer 0.5 seconde echte tijd en een heel klein beetje CPU-tijd. Als het programma in die 0.5 seconde niets anders uit te voeren heeft dan wordt de verwerking tijdelijk gestaakt, andere programma's krijgen voorrang, en de verwerking wordt pas weer voortgezet als de betreffende large page gearriveerd is. Een groot aantal large page faults vindt dan ook nauwelijks zijn weerslag in de CPU-tijd, maar is wel duidelijk te merken aan de totale verblijftijd van het programma in de computer. Om een redelijke 'turn around' tijd te krijgen doet men er dus verstandig aan de verwerking van data zo in te richten dat het aantal 'large page faults' zo beperkt mogelijk blijft. Ondanks dat dit door het systeem geregeld wordt, kan men hier wel degelijk invloed op uitoefenen.

Het voorbeeld zelf betreft de vermenigvuldiging van twee volle matrices C = AB. De orde van de vierkante matrices is 1024 en omdat er op een CYBER 205 large page 65 536 elementen kunnen worden opgeslagen, gaan er precies 64 kolommen van 1 matrix op één large page. De drie matrices beslaan dus in totaal 48 large pages, ofwel circa 3 miljoen woorden. Op een machine met een kerngeheugen van 1 miljoen woorden moet het grootste gedeelte dus op achtergrondgeheugen worden ondergebracht en resulteert het aanspreken van de achtereenvolgende kolommen van een matrix dus om de 16 kolommen in een large page fault.

We bekijken nu drie verschillende manieren om het matrix-matrixprodukt te berekenen en houden daarbij de analogie met de drie opeenvolgende BLAS-niveaus in het oog. We nemen aan dat er steeds ruimte is voor minstens 3 large pages in het geheugen, zodat van elk der betrokken matrices tenminste een gedeelte voorhanden kan zijn.

a. De klassieke berekening via inprodukten

Het FORTRAN-programmagedeelte hiervoor zou er als volgt kunnen uitzien (met N = 1024):

```
      DO 30 I = 1,N
        DO 20 J = 1,N
          DO 10 K = 1,N
            C(I,J) = A(I,K) * B(K,J) + C(I,J)
10        CONTINUE
20      CONTINUE
30    CONTINUE
```

We hadden al eerder gezien (paragraaf 7.3) dat deze berekeningswijze voor de CYBER 205, gelet op de verwerkingssnelheid, bepaald niet de meest optimale was vanwege de stride problemen in het doorlopen van de I-de rij van A in de DO 10-loop. De circa $2_{10}{}^{9}$ floating point operaties worden dus uitgevoerd met een scalaire snelheid van ongeveer 5 Mflops en dientengevolge bedraagt de CPU-tijd ongeveer 7 minuten. Dit getal verbleekt echter bij de I/O-tijd ten gevolge van page faults. Voor elke rij van A moeten 16 page faults gemaakt worden en omdat er ongeveer 10^6 inprodukten berekend moeten worden, komt dit neer op circa $16_{10}{}^{6}$ page faults à 1/2 seconde. De verblijftijd op de computer zou daarom minimaal *93 dagen* bedragen (als de computer continu aan dit programma zou kunnen werken). Merk echter op dat de klassieke inproduktsgewijze berekening geheel in stijl plaatsvindt volgens de BLAS-gedachte.

b. Kolomsgewijze bijwerking van C

Net zoals bij het matrixvectorprodukt (paragraaf 7.3) kunnen we de matrix-matrixproduktberekening ook kolomsgewijs inrichten door een simpele omwisseling van de loops. Bijwerking van de (eerder op nul geïnitialiseerde) matrix C vindt dan plaats volgens:

```
      DO 30 J = 1,N
        DO 20 K = 1,N
          DO 10 I = 1,N
            C(I,J) = A(I,K) * B(K,J) + C(I,J)
10        CONTINUE
20      CONTINUE
30    CONTINUE
```

De binnenste loop (DO 10) is nu een zogenaamde linked triad (of SAXPY), omdat B(K,J) een invariant is in deze loop. Op een 2-pipe CYBER 205 zal de verwerkingssnelheid hiervan dicht in de buurt van de 200 Mflops liggen, zodat de totale CPU-tijd voor de ongeveer 2_{10}^9 flops in de orde van een tiental seconden komt te liggen.

Voor iedere waarde van J moeten we alle kolommen van A aanspreken, dat vergt dus 1024 * 16 large page faults en dan zijn er nog hooguit 16 page faults nodig voor zowel B als C. Het totaal van ongeveer 16 000 page faults resulteert in een minimale I/O-tijd van ongeveer twee uur en een kwartier. Hoewel dit al veel gunstiger is als de onder a benodigde 93 dagen, overschaduwt de I/O-tijd nog steeds de CPU-tijd met vele orden van grootte.

We merken op dat de kolomsgewijze accumulatie van C, indien groepsgewijs uitgevoerd, weer goed aansluit bij de Level 2-BLAS; in feite bestaat de hier beschreven operatie uit N matrixvectoroperaties tussen de matrix A en de opeenvolgende kolommen van B.

c. Verdeling der matrices in submatrices

Dit idee is ook al uitvoerig beschreven in [44]. De matrices worden in dit geval onderverdeeld in vierkante blokken van 256 bij 256. Elk blok beslaat dan precies één large page. Dit vereist dus een geheel andere opzet voor de opslag der matrices en het leidt en passant ook tot kortere vectorlengten (namelijk 256 in plaats van 1024, echter dat is nauwelijks een bezwaar). We noteren de blokken met $\tilde{A}_{ij}$, $\tilde{B}_{ij}$ en $\tilde{C}_{ij}$ met $1 \le i,j \le 4$. Dus $\tilde{A}$, $\tilde{B}$ en $\tilde{C}$ zijn elk 4 bij 4 matrices waarvan de elementen zelf matrices zijn van 256 bij 256.

De bloksgewijze vermenigvuldiging komt er dan schematisch als volgt uit te zien:

```
for J = 1,2,3,4 do
    for K = 1,2,3,4 do
        for I = 1,2,3,4 do
            C̃ij = C̃ij + Ãik B̃kj
```

De I-loop bestaat uit de operatie matrix = matrix + matrix * matrix en kan op zich weer met de methode als onder b aangegeven worden uitgevoerd. Wegens de kortere looplengte zal de rekentijd nu een fractie meer gaan bedragen dan de onder b geschatte tien seconden, echter de grote winst ligt in een substantiële reductie van de I/O-tijd. De bovenstaande methode vergt slechts 144 large page faults en dat komt overeen met ongeveer 70 seconden I/O-tijd. Zelfs als de matrices oorspronkelijk op de klassieke

wijze (kolomsgewijs) stonden opgeborgen, dan is het nog bijzonder de moeite waard om de hele zaak bloksgewijs opnieuw op te bergen. Zoiets kan geschieden ten koste van ongeveer 100 large page faults.

We zien uit dit voorbeeld dat de bloksgewijze datastructuur voor matrices zich niet alleen uitstekend leent om een algoritme in meer samengestelde deeloperaties uit te schrijven (Level 3-BLAS), zodat efficiënter gebruik gemaakt kan worden van parallelle en vectormogelijkheden, maar dat het ook zeer goede mogelijkheden biedt voor een zuinig geheugenmanagement (reductie van transporten tussen verschillende niveaus van het geheugen).

ELF

MEERDERE PROCESSOREN; EEN INLEIDING

In de voorgaande hoofdstukken zijn we al diverse vormen van parallellisme tegengekomen. We zetten alles nog eens op een rijtje.

De meest eenvoudige vorm, en tot op heden ook de meeste succesvolle, is gebaseerd op segmentatie van de functional units zelf. Dit is een wel zeer fijnschalige variant van parallellisme, die alleen maar geschikt is voor het uitvoeren van een groot aantal gelijksoortige eenvoudige operaties achter elkaar volgens het principe van de lopende band. De micro-instructies worden dus wel parallel uitgevoerd, maar de verwerking van de opdrachten zelf wordt in feite alleen maar als een harmonica in elkaar gedrukt en het geheel houdt daarmee een serieel karakter. Anders gezegd, er vindt wel overlapping plaats, maar geen volledig parallellisme van de operaties.

Een verdere verfijning van het lopende-bandmechanisme wordt gerealiseerd door de mogelijkheid verschillende functional units achter elkaar te schakelen. Hierdoor kan bewerkstelligd worden dat er onder zekere gunstige voorwaarden simultaan kan worden gewerkt aan zowel de optelling als de vermenigvuldiging voor verschillende operandenparen.

Het segmentatieprincipe geeft dus aanleiding tot kleine lopende bandjes, de zogenaamde vector pipes en een volgende vorm van parallellisme wordt verkregen door het aanbrengen van meerdere vector pipes binnen een processor. De stroom operaties die aanvankelijk door één vector pipe (gesegmenteerde functional unit) geperst werd, kan nu verdeeld worden over meerdere (kortere) stromen. In dit geval worden de basisoperaties, als de optelling, dus echt parallel uitgevoerd. Echter nog steeds voeren de processoronderdelen niet simultaan verschillende (FORTRAN-)opdrachten uit.

Deze vorm van parallellisme, en haar verfijningen, wordt door Flynn [26] geclassificeerd in de rubriek SIMD: een enkele (vector)instructie volstaat voor uitvoering van een operatie op meerdere (paren) operanden. Onbesproken gebleven zijn hier SIMD-systemen waarbij een heel array van (zeer simpele) processoren gelijktijdig een zelfde

opdracht uitvoert (ICL-DAP) of waar zulks gebeurt door een hyperkubusstructuur van eenvoudige processoren (Connection Machine).

Het succes van het lopende-bandprincipe is gebaseerd op de mogelijkheid om bepaalde DO-loops, waarbinnen relatief eenvoudige opdrachten moeten worden uitgevoerd op array elementen, efficiënt te verwerken. De mogelijkheden en de beperkingen daarvan zijn in voorgaande hoofdstukken uitvoerig aan de orde geweest. Voor veel belangrijke technisch-wetenschappelijke toepassingen heeft vectorverwerking geleid tot een enorme schaalvergroting, anderzijds is voor heel wat toepassingen de snelheidswinst beperkt gebleven vanwege relatief te geringe vectorverwerkingsmogelijkheden binnen een gegeven algoritme (bijvoorbeeld vanwege scalair werk of vanwege zeer korte vectorlengten). Bovendien hebben we reeds gememoreerd dat zelfs de hoge maximale vectorsnelheden, die op dit moment gehaald kunnen worden, ontoereikend zijn voor het rekenen aan belangrijke modelleringsproblemen.

Allerwegen bestaat de indruk dat de sleutel tot verdere verlaging van de verblijftijd van een rekenprogramma op de computer gelegen is in een verdere benutting van parallellisme. Dit kan gerealiseerd worden door de centrale rekeneenheid (de processor) van de computer, al dan niet met een eigen geheugengedeelte, meervoudig aan te brengen. De hele computer bestaat dan uit een sterk gekoppeld stel min of meer zelfstandige computereenheden die elk stukken van een programma (of meestal ook complete zelfstandige programma's) onafhankelijk van elkaar kunnen uitvoeren. In de Flynn-classificatie brengt men dit soort computers onder in de MIMD-klasse. Om onderscheid te maken met computernetwerken, waarin verschillende (soorten) computers onderling verbonden zijn en met elkaar kunnen communiceren, worden de zelfstandige verwerkingseenheden van een parallelle computer aangeduid als processoren. In principe zou ook met behulp van een computernetwerk parallelle verwerking binnen een programma tot stand gebracht kunnen worden, echter gezien de vaak grote afstanden tussen de verbonden computers, hun geheel verschillende werking en snelheid, is dit niet praktisch aantrekkelijk.

Bij echte parallelle computers hebben we te maken met onderling nauw verbonden processoren, die allen van hetzelfde type zijn en die allemaal op dezelfde wijze geprogrammeerd kunnen worden. Voor deze processoren zijn communicatie- en synchronisatieregels opgesteld waardoor de hele zaak bestuurbaar wordt. In de meeste gevallen zijn de processoren onderling verwisselbaar, dus allemaal even snel en kan er dus van worden uitgegaan dat als ze simultaan en gelijktijdig aan rekenklussen van gelijke complexiteit gezet worden, zij daar allen ongeveer gelijktijdig mee klaar zullen zijn. Hierdoor is de programmeur in principe in de gelegenheid om het rekenwerk, voor zover het algoritme dat toelaat, evenredig over de processoren te verdelen waardoor het parallellisme optimaal benut kan worden.

Toen we de vectorcomputers de revu lieten passeren, kwamen we daar ook al machines met meerdere (vector)procesoren tegen, met name de CRAY X-MP met maximaal vier processoren, de IBM 3090 die met maximaal zes vectorprocessoren is uit te rusten en de ALLIANT FX/8 met maximaal acht vectorprocessoren.

De CRAY X-MP bestaat in maximale processorconfiguratie, grofweg gezegd, uit vier verbeterde CRAY-1 machines die nauw verbonden met elkaar in nagenoeg hetzelfde kabinet zijn geplaatst waarin vroeger één CRAY-1 geplaatst kon worden (in de loop

der tijd is de machine dus in wezen vier keer zo klein geworden). De vier processoren kunnen als afzonderlijke machines gebruikt worden en de algemene indruk is dat het meeste CRAY X-MP gebruik op die manier plaatsvindt. De capaciteit van de machine wordt dus in wezen benut om vier keer zoveel gebruikers te helpen als met één processor mogelijk was geweest. Voor zeer tijdkritische rekenproblemen, waaronder met name numerieke weersvoorspelling, waarbij het zaak is de voorspelling te kunnen doen ruim voordat de gebeurtenissen plaatsvinden, worden de processoren wel simultaan aan het werk gezet voor één enkel programma. Gaandeweg heeft de firma CRAY de mogelijkheden vergroot om op relatief simpele wijze binnen FORTRAN-programma's de beschikbare capaciteit van eventuele vrije processoren te benutten. We zullen hier nog uitgebreider op terugkomen.

Van de IBM 3090 bestaan inmiddels exemplaren die met meer VF's (aankoppelbare vectorprocessoren) zijn uitgerust. Het is ons niet bekend of er momenteel al op wat grotere schaal van die simultane parallelle verwerkingsmogelijkheden gebruik gemaakt wordt. Dongarra [17] maakt gewag van snelheidsmetingen die verricht zijn op een systeem met zes parallelle VF-eenheden. Op dit moment ontbreekt eigen praktische ervaring om hier dieper op in te kunnen gaan.

Ook voor de ALLIANT FX/8 was er een gegronde reden om deze machine in eerste instantie onder te brengen in de galerij der vectorcomputers. In de meest eenvoudige vorm bestaat de machine uit slechts één enkele (vector)processor en dan is er dus geen wezenlijk verschil met de overige vectorcomputers. Machines die zijn uitgerust met meerdere processoren, tot maximaal acht, kunnen net als bij de CRAY X-MP benut worden om er meerdere programma's tegelijk op te laten verwerken om zo de doorstroomcapaciteit van de machine te vergroten. Wanneer er meerdere processoren ingezet worden bij de verwerking van één enkel programma dan gebeurt dat op een manier die voor de oppervlakkige beschouwer grote gelijkenis vertoont met de verwerking van programma's op vectorcomputers met meerdere vector pipes. De verschillende processoren worden niet afzonderlijk door de programmeur van instructies voorzien, het systeem draagt zelf zorg voor het automatisch parallel verwerken van programmaonderdelen op DO-loop-niveau. Voor eenvoudige loops, die bijvoorbeeld zelf al vectoriseerbaar waren, komt dit neer op een splitsing van de loop over de processoren en dat lijkt dan op de werking van de al eerder besproken meer-pipe's-systemen, zoals de CYBER 205 en de Japanse machines. Echter ook samengestelde loops en loops die een seriële code bevatten kunnen worden opgesplitst en zelfs loops waarbij de code voor de verschillende loop-doorgangen verschillend is. Er ontstaat zo dus echte MIMD-verwerking. We zullen hier ook nog voorbeelden van geven. Niet onvermeld mag blijven dat de ALLIANT FX/8-computers zelf weer als bouwstenen fungeren in het zogeheten CEDAR-project dat uitgevoerd wordt onder leiding van professor Kuck aan de Universiteit van Illinois.

Het idee op zich klinkt zeker simpel, maar het construeren van praktisch zinvolle systemen die uit meerdere processoren bestaan is allerminst eenvoudig. Het eerste probleem dat zich aan ons voordoet is dat men niet onbeperkt veel processoren op hetzelfde centrale geheugen kan aansluiten en zelfs ook kleine aantallen zorgen al voor talloze problemen. Bij krachtige (vector)processoren bedraagt het aantal rechtstreeks op een centraal geheugen aan te sluiten exemplaren een stuk of acht. Dat is tenminste wat we

gerealiseerd zien bij de huidige systemen alsook bij de systemen die zijn aangekondigd voor de zeer nabije toekomst: de CRAY X-MP, CRAY-2 en CONVEX-200 serie met vier processoren, de IBM 3090 met 6 VF's, de (minisuper) ALLIANT FX/8, de CRAY Y-MP (de gedoodverfde opvolger van de CRAY X-MP) en de ETA 10 (de opvolger van de CYBER 205) met maximaal acht processoren.

Bij minder krachtige processoren, zoals bij de SEQUENT machines, kunnen meerdere processoren, bijvoorbeeld 30, op een centraal geheugen worden aangesloten, al gebeurt dat in werkelijkheid veelal ook indirect via een tussengeheugen. Dit kan onder andere doordat de processoren zelf veel minder snel zijn waardoor de totale toegangscapaciteit van het geheugen, de zogenaamde bandbreedte, benut kan worden door meerdere processoren. Het lijkt dan vanuit gebruikersstandpunt gezien alsof alle processoren rechtstreeks aan hetzelfde centrale geheugen hangen, maar, zoals gezegd, dit hoeft fysiek niet het geval te zijn. Echter ook in deze gevallen is het maximaal aantal processoren beperkt.

Een computersysteem waarbij de processoren hun data uit een gemeenschappelijk geheugen putten duidt men aan met de term shared memory systeem. Hoewel dit systeem in principe eenvoudig lijkt te besturen (we hoeven ons immers niet bezig te houden met de indeling van de data over lokale geheugens), kan het in de praktijk toch aanleiding geven tot veel moeilijkheden. Doordat de processoren in principe onafhankelijk en niet synchroon werken, kan het gebeuren dat er data uit het geheugen worden gehaald die op dat moment nog niet de juiste waarde hebben omdat de processor die ze had moeten genereren wat vertraging heeft ondervonden. Verder kan het voorkomen dat processoren op hetzelfde moment data naar dezelfde plaats in het geheugen sturen of aldaar data opvragen. Door dit soort, meestal onbedoelde, gebeurtenissen kan het voorkomen dat herhaalde uitvoering van eenzelfde programma op identieke data aanleiding geeft tot verschillende resultaten, domweg omdat er verschillen in de volgorde van databehandeling kunnen ontstaan. Uiteraard is dat een gevolg van foute of onzorgvuldige programmering waarbij de deelklussen niet goed op elkaar zijn afgestemd. Het is echter verdraaid lastig om dit soort programmeerfouten in de praktijk op te sporen (vooral omdat een gesignaleerde fout bij een tweede keer draaien niet hoeft voor te komen). Een ander probleem is dat de processoren weliswaar niet gelijkertijd naar dezelfde geheugenplaatsen refereren, maar wel op hetzelfde moment toegang eisen tot dezelfde geheugenbank. Dit leidt dan tot geheugenbankconflicten die de gebruiker niet altijd zelf kan vermijden. Aan het bedrijfssysteem van dit soort shared memory systemen worden hoge eisen gesteld om de nadelige effecten van allerlei synchronisatieproblemen te beperken.

Een andere weg die men gekozen heeft om de geheugenproblematiek te omzeilen en om het aantal processoren verder te kunnen opvoeren is dat men elke processor uitrust met zijn eigen lokale geheugen. De processoren werken nu in principe ongestoord op data die zich in het eigen lokale geheugen bevinden en als er voor verdere verwerking (tussen)resultaten van een andere processor nodig zijn, dan moet deze processor die data expliciet naar de vragende processor zenden, waarna de vragende processor na ontvangst de verwerking kan voortzetten. Men spreekt vrij algemeen van *message passing* systemen of *local memory* systemen. Dit soort systemen kent dus geen gemeenschappelijk centraal geheugen (hooguit een gemeenschappelijk achtergrondgeheugen, zoals schijven of tapes). De processoren zitten onderling in een communicatieweb met elkaar

verbonden en worden van data en instructies voorzien door een *host computer*. Het geheugen van deze host kan men ook niet zien als een soort centraal geheugen omdat het benutten van gegevens hieruit ook weer via expliciete zendopdrachten en ontvangstopdrachten moet geschieden.

Het aantal processoren dat men direct kan verbinden met één enkele processor is momenteel ook nog zeer beperkt. Men moet daarom kiezen voor een onderlinge verbindingsstructuur en dit heeft geleid tot veel uiteenlopende computerarchitecturen. Er schijnt zich tegenwoordig een zekere voorkeur te ontwikkelen voor de zogenaamde hyperkubusconfiguraties, zeker waar het gaat om grote aantallen processoren. Verschillende commercieel verkrijgbare systemen zijn volgens het hyperkubusprincipe opgezet. We zullen nog uitgebreider op de hyperkubusstructuur terugkomen.

De message passing of local memory systemen hebben weer hun eigen specifieke problemen. Het blijkt dat het overzenden van data van de ene naar de andere processor een relatief kostbare zaak is ten opzichte van de berekeningen zelf. Ervaringen met de huidige systemen wijzen erop dat men vele honderden (eenvoudige) rekenoperaties parallel moet kunnen uitvoeren om het vertragend effect van het overzenden van een enkel geheugenwoord te verdoezelen [45]. Dit hangt natuurlijk sterk af van de verhouding tussen de communicatiesnelheid en de rekensnelheid en kan dus door veranderingen in de techniek in de toekomst geheel anders uitpakken.

Het kan ook een wezenlijk probleem zijn dat lang niet alle processoren onderling direct met elkaar verbonden zijn, hetgeen inhoudt dat het overzenden van data tussen twee processoren vaak via tussenstations (dat wil zeggen: andere processoren) moet geschieden. Dit wordt meestal door het systeem zelf geregeld, dus de programmeur hoeft zich niet bezig te houden met het zoeken naar geschikte tussenstations. Hij moet zich er echter wel terdege van bewust zijn dat dit achter de schermen gebeurt. Verbindingslijnen tussen twee naburige processoren kunnen dus belast zijn met het overzenden van data tussen twee geheel andere processoren en dit kan aanleiding geven tot voorrangs- en verstoppingsproblemen. Hoewel dit soort problemen meestal voor de gebruiker wordt afgeschermd en min of meer optimaal door het systeem zelf wordt opgelost, zal het wel duidelijk zijn dat we de programmering van een parallel algoritme niet zo moeten inrichten dat er hoofdzakelijk communicatie is tussen de verst van elkaar verwijderde processoren. Een aantrekkelijk computersysteem om processorverbindingen te bestuderen wordt geleverd door MEIKO. Bij deze parallelle computers kan de gebruiker binnen zekere grenzen zelf het verbindingsnetwerk definiëren.

Omdat de processoren alleen toegang hebben tot hun eigen geheugen en uitwisseling alleen plaatsvindt wanneer één processor expliciet geïnstrueerd wordt bepaalde data te verzenden en een andere eveneens geïnstrueerd wordt die data te accepteren, heeft men geen last van het feit dat processoren onbedoeld (elkaars) data kunnen 'verzieken' door slecht gesynchroniseerde acties, zoals dat kon gebeuren bij shared memory systemen. Mede hierdoor zijn codes voor message passing systemen meestal eenvoudiger (alles is relatief!) foutvrij te krijgen. De shared memory systemen stellen evenwel minder eisen aan de datastructuur, en dus aan de structuur van een algoritme, en zijn daarom in het algemeen wat eenvoudiger programmeerbaar. Over dit soort en andere (vermeende) voor- en nadelen zullen we in het vervolg van dit boek, waar het zo uitkomt, nog verder filosoferen.

Voordat we ons wagen aan een voorzichtige bespreking van een aantal systemen lijkt het goed nog eens in te gaan op de beperkingen en doeleinden van die bespreking.

Er zijn en worden vele soorten parallelle computers ontworpen en gebouwd over de gehele wereld. De meeste daarvan zien het licht in een typische onderzoekomgeving en het leeuwedeel van de parallelle machines wordt ook gebruikt om parallelle verwerkingsmogelijkheden verder te onderzoeken. Slechts enkele ontwerpen hebben het vooralsnog gebracht tot commerciële serieproduktie. Het is nog niet duidelijk welke architecturen op den duur de beste mogelijkheden zulllen kunnen gaan bieden.

Met enkele van deze machines heeft de auteur enige ervaring opgedaan. Van sommige machines is inmiddels het een en ander uit de literatuur bekend. Getracht is om een representatief overzicht te bieden van exploitabele parallelle verwerkingsmogelijkheden, steunend op eigen praktijkervaring en op betrouwbare literatuurgegevens. Het noemen van bepaalde merken houdt dan ook niet meer in dan dat er een concreet voorbeeld van een zekere architectuur wordt aangegeven, het niet noemen van alternatieven betekent slechts dat de auteur zich onvoldoende op de hoogte voelde om daarover zinvol te berichten.

Een andere reden om een vorm van volledigheid niet na te streven is gelegen in de observatie dat er vrijwel maandelijks computermerken verschijnen en andere weer verdwijnen. De reden voor het verdwijnen van een fabriek van de markt is niet altijd gelegen in een verkeerde ontwerpkeuze, maar heeft meer te maken met ontwikkelingskosten en vertrouwen van de beleggers. Zo heeft bijvoorbeeld de parallelle DENELCOR HEP computer een behoorlijke invloed gehad op de ontwikkeling van het parallelle onderzoek; de firma DENELCOR zag evenwel geen kans haar geavanceerde ideeën tijdig en goed te realiseren en de firma ging in 1985 failliet. Nog steeds zijn er exemplaren van de HEP in gebruik, onder andere bij de Messerschmidt fabrieken in West Duitsland.

In [16] wordt een up to date overzicht gegeven van commercieel beschikbare parallelle en vectorcomputers. Het overzicht is evenwel zeer incompleet voorwat betreft specifieke gegevens over deze computers (zoals bijvoorbeeld prestatievermogenparameters).

Op vele plaatsen wordt hard en intelligent gewerkt aan de ontwikkeling van parallelle machines die geschikt zijn voor het oplossen van een specifieke klasse problemen. Een voorbeeld hiervan is het werk aan de Technische Universiteit van Delft waar, onder leiding van de hoogleraren Hogendoorn en Veltman, aan een machine speciaal voor het oplossen van bepaalde Navier-Stokes vergelijkingen wordt gewerkt. Ook dit soort machines blijven in dit boek buiten nadere bespreking. We richten de aandacht uitsluitend op machines die bedoeld zijn voor vrij algemene technisch wetenschappelijke toepassingen en dan bovendien nog programmeerbaar zijn in een algemene programmeertaal, en meer in het bijzonder FORTRAN (inclusief lokale aanpassingen en uitbreidingen). Wij hopen met deze keuze de lezer enigszins de helpende hand te reiken bij het vinden van een weg in dit nog pas nauwelijks tot ontwikkeling gekomen gebied.

TWAALF

VERWACHTINGEN TEN AANZIEN VAN PARALLEL REKENEN

12.1 Enige bespiegelingen en meningen vooraf

De verwachtingen ten aanzien van de mogelijkheden van parallel rekenen lopen nogal uiteen. Het onderzoek naar die mogelijkheden bevindt zich momenteel in een verkennend stadium en het is nog allerminst duidelijk wat de meest geschikte computer architecturen zijn of de meest flexibele en geschikte programmeertalen (flexibel in de zin dat vele vormen van parallellisme eenvoudig en herkenbaar tot uitdrukking kunnen worden gebracht).

Een heel scala van mogelijkheden wordt momenteel geboden door uiteenlopende computermerken. Er is inmiddels overtuigend aangetoond dat bepaalde vormen van parallelle verwerking de grenzen van het haalbare een flink eind verder kunnen leggen. Het is van groot belang de ontwikkelingen op de voet te blijven volgen als men geen achterstand wenst op te lopen op die gebieden waar het grootschalig rekenwerk een steeds belangrijker plaats inneemt. Het volgen van verschillende mogelijkheden, in plaats van af te wachten welke vorm de winnaar zal blijken, is nodig omdat het waarschijnlijk zo zal zijn dat er in de naaste toekomst verschillende vormen van parallelle verwerking naast elkaar bestaansrecht zullen hebben. Naast de technische mogelijkheden spelen daarbij natuurlijk ook de kosten van zowel hardware als software een belangrijke rol, alsmede het soort toepassingen dat men voor ogen heeft.

Afgezien van zeer specifieke toepassingen die uitermate parallel van aard zijn, zoals bijvoorbeeld verwerking van signalen (seismologische toepassingen) en digitale beeldverwerking, lijkt enige reserve ten aanzien van het welslagen en het resultaat van parallel rekenen (in de vorm van een reductie van de tijd die verstrijkt voordat de computer met het probleem klaar is) ook op zijn plaats. Voor een daadwerkelijke doorbraak van het werken met enige tientallen processoren is nog technische vooruitgang vereist

alsmede een reductie van de kosten. Niet in de laatste plaats wordt dit soort massief of grootschalig parallellisme pas echt rendabel, in die zin dat p processoren ook tot een O(p) reductie in de reële verwerkingstijd leiden, indien het toegepast kan worden bij zeer grote rekenproblemen. Waarschijnlijk zal ook hier weer door een wisselwerking tussen technische mogelijkheden en probleemaanpak op den duur duidelijk worden welk soort machine het best geëigend is voor een bepaalde werkomgeving. We zullen de verwachtingen die men überhaupt mag koesteren ten aanzien van het parallelle rekenen in de volgende paragraaf wat beter onderbouwen, met behulp van de wet van Amdahl.

Voor geschikte modelproblemen zijn inmiddels al veelbelovende resultaten behaald, echter voor de meeste echte grote rekenproblemen die zich voordoen in industriële en wetenschappelijke toepassingen zijn de ervaringen over het algemeen nog niet zo gunstig. Dit wordt wellicht nog het best geïllustreerd door een prijsvraag die wiskundige en software expert Alan Karp in oktober 1985 uitschreef [37]. Men moet uit het uitschrijven van de prijsvraag niet afleiden dat parallel rekenen slechts beperkte mogelijkheden biedt. De reserves die uit Karps formuleringen blijken, zijn voornamelijk gericht op de beperkte verwachtingen (vooralsnog) ten aanzien van de zogenaamde pre- en post-processing. De letterlijke tekst van de prijsvraag luidt als volgt:

A PARALLEL PROCESSING CHALLENGE

by
Alan Karp
2227 Tasso Street
Palo Alto, CA 94301, USA
na.karp@su-score.arpa

I have just returned from the Second SIAM Conference on Parallel Processing for Scientific Computing in Norfolk, Virginia. There I heard about 1,000 processor systems, 4,000 processor systems, and even a proposed 1,000,000 processor system. Since I wonder if such systems are the best way to do general purpose, scientific computing, I am making the following offer:

> I will pay $100 to the first person to demonstrate a speed-up of at least 200 on a general purpose, MIMD computer used for scientific computing. This offer will be withdrawn at 11:59 p.m. on December 31, 1995.

Some definitions are in order:

> *Speed-up*
>
> The time taken to run an application on one processor using the best

sequential algorithm divided by the time to run the same application on N processors using the best parallel algorithm.

General Purpose

The parallel system should be able to run a wide variety of applications. For the purposes of this test, the machine must run three different programs that demonstrate its ability to solve different problems. I suggest a large, dynamic structures calculation, a trans-sonic fluid flow past a complex barrier, and a large problem in econometric modelling. These are only suggestions; I will be quite flexible in the choice of the problems. Several people have volunteered to adjudicate disputes in the selections of the applications.

Application

The problems run for this test must be complete applications, no computational kernels allowed. They must contain all input, data transfers from host to parallel processors, and all output. The problems chosen should be the kind of job that a working scientist or engineer would submit as a batch job to a large supercomputer. In addition, I am arbitrarily disqualifying all problems that Cleve Moler calls 'embarrassingly parallel'. These include signal processing with multiple independent data streams, the computation of the Mandelbrot set, etc..

There are some rather obvious ground rules:

Simulations of the hardware are not permitted. I am looking for a demonstration on a running piece of hardware.

The same problem should be run on the sequential and parallel processors.

It is not fair to cripple the sequential processor. For example, if your operating system uses 99% of the processor, the single processor run will spend all its time in the operating system. Super-linear speed-up as the number of processors is increased is evidence of this problem.

It is not fair to memory starve the single processor run. If you have 100K words of memory on each of 1,000 processors, and you run an 10 MW problem, it is not fair for the sequential run to be made on a 100 KW processor. After all, we are not interested in seeing the impact of additional memory; we want to see how much the extra CPUs help.

It may not be possible to follow all these additional rules. For example, you

cannot be expected to build a single processor with lots of extra memory or write a new operating system just for this test. However, you should do your best to make the demonstration fair. A third party has agreed to adjudicate any scaling disputes.

Anyone claiming this prize will be expected to present his or her results at a suitable conference. At the end of the presentation I will have the audience vote on whether the demonstration was successful. Remember, the purpose of the bet is to force developers to demonstrate the utility of massively parallel systems, not to show they can find holes in the rules.

[Nadat de tekst van dit boek voltooid was, kwam ons ter ore dat de prijs inmiddels is toegekend aan een een groep personen van Sandia Labs. te Livermore, USA. Deze groep heeft versnellingsfactoren van 400-600 gerealiseerd voor drie verschillende toepassingen. Bijzonderheden hierover worden vermeld in het tijdschrift IEEE Software (mei 1988) en in het juli 1988-nummer van SIAM J.Sci.Stat.Comput.]

Het Los Alamos National Laboratory ligt in de frontlijn waar het gaat om het inpassen van geavanceerde computers bij het grootschalig rekenwerk. Men beschikt er in ruime mate over allerlei soorten vector en parallelle rekenautomaten. Op grond van ervaringen en gericht onderzoek concludeerde Bucher [2] in 1983 dat het onwaarschijnlijk geacht moest worden dat techieken als bijvoorbeeld de Josephson Junction tot de gewenste vergroting van de rekensnelheden zou leiden binnen het eerstvolgende tiental jaren. Men mag volgens haar veilig aannemen dat die versnelling wel gerealiseerd zal worden door meer parallellisme te introduceren, waarschijnlijk in de vorm van meerdere scalaire of vectorprocessoren die asynchroon werkend op één gemeenschappelijk geheugen zijn aangesloten. Communicatiekosten en de implicaties van Amdahls wet (zie hierna) zullen het aantal zinvol samenwerkende processoren (voor algemene toepassingen!) beperken tot waarschijnlijk niet meer dan 16.

Niettegenstaande de door Karp (impliciet) en Bucher (expliciet) verwoorde bedenkingen lijkt de computerindustrie toch onstuitbaar de weg naar grotere aantallen processoren te zijn ingeslagen. De bekende computerontwerper Seymour Cray, die zich ooit zeer terughoudend opstelde tegenover het ontwerpen van (super)computers met meer dan vier processoren, kondigde in november 1987 de ontwikkeling van een zeer krachtig uitgerust 64-processor systeem aan [6]. We zullen hier later nog op terugkomen.

12.2 Parallellisme en verwerkingstijd

Bij parallelle verwerking vergelijkt men de rekentijd op p processoren met die welke gemeten wordt op één enkele processor om zo een indruk te krijgen van de geboekte winst. Als de verwerkingstijd daardoor niet met een factor evenredig aan p korter is

geworden, dan kan men zich afvragen of het niet aantrekkelijker is om de processoren onafhankelijk te benutten voor geheel verschillende klussen. Immers als een bepaalde rekenpartij één uur duurt en bijvoorbeeld bij parallelle verwerkingen door vier processoren een half uur, dan kan men dus in één uur ofwel twee problemen parallel doorrekenen, ofwel vier problemen niet parallel.

Het zou ideaal zijn als het proces op p processoren ook nagenoeg p keer zo snel klaar zou zijn als op één processor, echter precies zoals bij vectorprocessing gelden er hier ook stringente beperkingen op de snelheidswinst als gevolg van niet-parallel verwerkbare gedeelten van het rekenproces. Ook hier geldt weer, geheel in de stijl van de wet van Amdahl, dat als 10% van de operaties in een rekenpartij niet-parallel kunnen worden uitgevoerd, de totale snelheidswinst hooguit een factor 10 kan bedragen ongeacht het aantal processoren dat (parallel) wordt ingezet om het parallelle gedeelte, ter grootte van 90%, te verwerken.

We gaan dit weer wat preciezer analyseren en bekijken eerst maar eens een gegeven vast proces dat n operaties omvat. Met T_p geven we de rekentijd aan die nodig is om het hele proces zo goed mogelijk parallel uit te voeren met p processoren.

De tijdwinst (*speed-up*) S_p wordt gedefinieerd als

$$S_p = \frac{T_1}{T_p}$$

Deze factor S_p geeft dus aan hoeveel keren sneller het gegeven rekenproces klaar is wanneer er p processoren ingezet worden in plaats van één. Verder definiëren we nog de *effectiviteit* E_p als

$$E_p = \frac{S_p}{p},$$

dat wil zeggen de snelheidswinst per processor. E_p geeft in feite aan hoe effectief elk der p processoren benut wordt in vergelijking met verwerking van het hele proces op één processor. Als E_p klein is ten opzichte van 1, hadden we misschien beter p verschillende programma's op de p processoren kunnen laten verwerken.

Laten we nu eens aannemen dat een fractie f van de operaties parallel verwerkt kan worden en dat de rest, dat zijn dus (1-f)n operaties, slechts met één processor kan worden uitgevoerd. De benodigde rekentijd voor dit niet-parallelle gedeelte is dan

$$\frac{(1-f)n}{n} T_1 = (1-f)T_1 .$$

Als het parallelle gedeelte inderdaad parallel door p processoren verwerkt kan worden, dan is, afgezien van allerlei synchronisatiekosten, de rekentijd daarvoor:

$$\frac{fn}{n} \frac{T_1}{p} = \frac{f}{p} T_1$$

De totale rekentijd bij parallelle verwerking is

$$T_p = \frac{f}{p} T_1 + (1-f)T_1$$

en dientengevolge bedraagt de snelheidswinst

$$S_p = \frac{T_1}{T_p} = \frac{p}{f + (1-f)p} \leq \frac{1}{1-f},$$

en de effectiviteit

$$E_p = \frac{1}{f + (1-f)p}.$$

We plaatsen hier alvast enkele kanttekeningen bij.

a. Het aantal operaties moet natuurlijk wel groot genoeg zijn om de p processoren zinvol aan het werk te houden. Als er n operaties moeten worden uitgevoerd dan houdt dat in de praktijk veelal in dat p veel kleiner moet zijn dan n.

b. Het door ons beschouwde snelheidsmodel stelt de zaken wat te eenvoudig voor. In het algemeen leidt het parallel rekenen zelf ook tot wat extra rekenwerk, nodig voor distributie van de opdrachten (en/of data) over de processoren en voor de onderlinge afstemming van de werkzaamheden (synchronisatie). Deze extra kosten moeten in feite nog in T_p verrekend worden (waardoor T_p dus groter wordt) en dat leidt in de praktijk meestal tot een merkbaar lagere S_p dan die welke door ons model voorspeld wordt.

c. Lang niet elk rekenschema leent zich zonder meer tot parallelle verwerking. Dat houdt in dat men (delen van) het rekenschema vaak moet vervangen door beter parallelliseerbare processen. Deze processen vergen meestal meer operaties (als dat niet zo was dan zouden ze ook op seriële computers al gebruikt zijn), en dan is het dus niet erg eerlijk om de rekentijd voor het nieuwe proces op p processoren te vergelijken met de rekentijd voor hetzelfde proces op één processor. Als men slechts één processor zou hebben dan zou men uiteraard voor het oude proces gekozen hebben en het is dus meestal zinvoller om de rekentijd van het nieuwe proces te vergelijken met die van het oude proces op één processor. Ook dit leidt in de praktijk vaak tot een verdere verlaging van S_p.

Ondanks deze waslijst aan bezwaren is het model toch nuttig, omdat het ons althans bovengrenzen verschaft voor de te verwachten snelheidswinst en dat kan ons helpen al te overspannen verwachtingen te temperen.

Laten we bijvoorbeeld eens aannemen dat voor een gegeven probleem bij verwerking op vier processoren een snelheidswinst van 3.7 gemeten wordt, dus $S_4 = 3.7$ en $E_4 = 0.925$ (lang niet gek niet dus).

Dit ziet er bemoedigend uit, echter alvorens bij de leveranciers een order te plaatsen voor nog maar eens 4 * zoveel processoren (in de hoop dan weer een snelheidswinst 3.7 extra, dus 3.7 * 3.7 = 13.7, te behalen), kijken we eerst eens naar de voorspelde *bovengrenzen* voor S_p volgens ons model.

Uit $S_4 = 3.7$ volgt

$$\frac{4}{f + (1-f)4} = 3.7 \quad \text{en dus } f = 0.973.$$

Hieruit leidt men eenvoudig de volgende waarden af:

$S_{16} = 11.4$ (in plaats van de gehoopte 13.7), $E_{16} = 0.71$
$S_{64} = 23.7$, $E_{64} = 0.37$
$S_{1024} = 35.7$, $E_{1024} = 0.035$

en tenslotte

$S_{\infty} = 37$, $E_{\infty} = 0$ (heel droevig dus).

We hadden de bui al kunnen zien aankomen, immers

$$S_p \leq \frac{1}{1-f}$$, voor alle waarden van p.

We zien uit dit voorbeeld dat de winst al betrekkelijk snel wordt afgevlakt: met vier processoren boekten we een winst van 3.7, door nog eens vier keer zoveel processoren in te schakelen wordt dat (hooguit) 11.4 (in plaats van 13.7) en bij bijvoorbeeld 1024 processoren wordt de winst maximaal 35.7 (in plaats van $3.7^5 \approx 693$). De p processoren worden in vergelijking met één processor steeds minder effectief benut. In de volgende tabel staan waarden van S_p voor diverse groottes van de parallelliseerbare fractie f van een rekenpartij en men dient zich hierbij terdege te realiseren dat de gemeten waarden daar in de praktijk nog onder liggen.

f	p=1	p=2	p=3	p=4	p=8	p=16	p=32	p=64	p=oneindig
1.00	1.00	2.00	3.00	4.00	8.00	16.00	32.00	64.00	oneindig
.99	1.00	1.98	2.94	3.88	7.48	13.91	24.43	39.26	100.00
.98	1.00	1.96	2.88	3.77	7.02	12.31	19.75	28.32	50.00
.97	1.00	1.94	2.83	3.67	6.61	11.03	16.58	22.14	33.33
.96	1.00	1.92	2.78	3.57	6.25	10.00	14.29	18.18	25.00
.95	1.00	1.90	2.73	3.48	5.93	9.14	12.55	15.42	20.00
.94	1.00	1.89	2.68	3.39	5.63	8.42	11.19	13.39	16.67
.93	1.00	1.87	2.63	3.31	5.37	7.80	10.09	11.83	14.28
.92	1.00	1.85	2.59	3.23	5.13	7.27	9.19	10.60	12.50
.91	1.00	1.83	2.54	3.15	4.91	6.81	8.44	9.59	11.11
.90	1.00	1.82	2.50	3.08	4.71	6.40	7.80	8.77	10.00
.75	1.00	1.60	2.00	2.28	2.91	3.37	3.66	3.82	4.00
.50	1.00	1.33	1.50	1.60	1.78	1.88	1.94	1.97	2.00
.25	1.00	1.14	1.20	1.23	1.28	1.31	1.32	1.33	1.33
.10	1.00	1.05	1.07	1.08	1.09	1.10	1.11	1.11	1.11
.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Tabel 2. Versnellingsfactoren S_p.

We hebben nu bekeken wat er gebeurt als we voor een vaste rekenpartij het aantal processoren opvoeren. Als men echter eenmaal de beschikking heeft over een machine met p processoren dan kan men zich ook afvragen wat het effect is als men grotere problemen verwerkt (grotere waarden van n).

Moler [47] voert hiervoor het begrip *effectief parallel algoritme* in. We noteren daarvoor eerst de niet-parallelliseerbare fractie van het totale aantal operaties als α (n) = 1 - f. Merk op dat we de fractie nu van de probleemgrootte n laten afhangen. Moler claimt namelijk dat voor veel algoritmen deze fractie α (n) afneemt wanneer we de probleemgrootte n opvoeren (bij een vast aantal processoren!), dat wil zeggen dat dan een relatief steeds kleiner gedeelte niet-parallelliseerbaar is. Hij definieert daarom een effectief parallel algoritme als een algoritme waarvoor geldt:

$$\alpha(n) \longrightarrow 0 \text{ als } n \longrightarrow \infty$$

Voor zo'n algoritme geldt dan bij gebruik van p_0 processoren:

$$S_{P_0} = \frac{p_0}{1 + (p_0-1)\alpha(n)} \text{ en } E_{P_0} = \frac{1}{1 + (p_0-1)\,\alpha(n)}$$

en we zien dat, door n op te voeren, de snelheidswinst uiteindelijk op het gewenste niveau komt:

$$S_{P_0} \longrightarrow p_0$$
$$E_{P_0} \longrightarrow 1 \qquad \text{als } n \longrightarrow \infty.$$

Door het opvoeren van de probleemgrootte krijgen we dus steeds meer een S_p die min of meer lineair met p oploopt, voor $p \leq p_0$, en de processoren worden gaandeweg steeds *effectiever* benut.

Echter door vervolgens p verder toe te laten nemen neemt de versnellingsfactor S_p niet langer lineair toe en zakt ook de effectiviteit. In feite zien we dat bij een vast (groot) aantal processoren de vraag rijst of de geheugens van de processoren (of het totale kerngeheugen) wel groot genoeg zijn om de probleemgrootte aan te kunnen waarvoor S_{P_0} in de buurt van p_0 zou komen. Moler [47] merkt op dat deze geheugengrootte in feite een grotere beperking is bij het realiseren van grote versnellingsfactoren dan het aantal processoren: men kan vaak simpelweg een probleem niet zo groot maken dat de potentiële snelheidswinst p_0 te realiseren is.

We besluiten met de resultaten van een analyse van Stone [54]. Volgens deze analyse zijn de volgende snelheidswinsten althans theoretisch realiseerbaar (mits de probleemgrootte n >> p):

PROBLEEM	VERSNELLING
volle matrixproblemen; eindige elementenproblemen	$\sim p$
sorteren, tridiagonale stelsels, polynoomevaluatie	$\sim \frac{p}{\log p}$
zoekprocessen	$\sim \log p$
niet-lineaire recursies	vrijwel geen versnelling

Tabel 3. Mogelijke snelheidswinsten bij parallelle verwerking.

12.3 Synchronisatiekosten en effectiviteitsverlies

In paragraaf 12.2 hebben we wat gefilosofeerd over de invloed van het niet-parallelle rekengedeelte van een algoritme op de uiteindelijke snelheid. In deze paragraaf zullen we proberen de rekentijd en het prestatievermogen van een parallelle computer te vangen in termen van een paar karakteristieke parameters, geheel in de stijl van de paragrafen 6.1 en 6.2. We richten daarbij de aandacht uitsluitend op het parallel uit te voeren rekengedeelte.

Als model beschouwen we een parallelliseerbaar proces A dat bestaat uit N gelijke deelprocessen die elk serieel moeten worden uitgevoerd, waarbij de verwerkingstijd van zo'n deelproces op één processor t_s sec. vergt. Wanneer deze N deelprocessen onafhankelijk, dat wil zeggen parallel, worden uitgevoerd door p processoren, dan kunnen we de rekentijd die verstrijkt tussen het begin en het eind van het hele proces schrijven als

$$t_p = \left\lceil \frac{N}{p} \right\rceil t_s + t_{s,p}$$

Met $\lceil x \rceil$ geven we, voor $x \geq 0$, afronding naar boven aan. De grootheid $t_{s,p}$ representeert de extra tijd, uitgedrukt in μsec, die bij het parallel uitvoeren verloren gaat ten gevolge van synchronisatiekosten (het werk moet tenslotte verdeeld worden over de processoren en er moet gecontroleerd worden of de processoren klaar zijn).

Wanneer elk deelproces k_d flops omvat, dan is de rekensnelheid van het gehele proces A op één processor gelijk aan

$$R(A,1) = \frac{Nk_d}{Nt_s} = \frac{k_d}{t_s} \text{ Mflops.}$$

De tijd t_p voor p processoren kunnen we nu schrijven als

$$t_p = \left\lceil \frac{N}{p} \right\rceil \frac{t_s}{k_d} k_d + t_{s,p} \approx \frac{1}{pR(A,1)} (Nk_d + pR(A,1)t_{s,p})$$

waarbij we voor $\lceil \frac{N}{p} \rceil$ hebben geschreven $\lceil \frac{N}{p} \rceil \approx \frac{N}{p}$.

De factor pR(A,1) geeft de verwerkingssnelheid van de gezamenlijke p parallelle processoren weer, afgezien van de synchronisatiekosten. Dus de term $pR(A,1)t_{s,p}$ geeft aan hoeveel operaties (in eenheden van een miljoen floating point operaties) er door de p processoren in totaal parallel hadden kunnen worden uitgevoerd in de nu aan synchronisatie bestede tijd.

Voor de rekensnelheid van het p-processorsysteem geldt

$$R(A,p) = \frac{Nk_d}{\frac{1}{pR(A,1)} (Nk_d + pR(A,1)t_{s,p})}$$

$$= pR(A,1) \frac{1}{1 + \frac{pR(A,1)t_{s,p}}{Nk_d}}$$

Als N zeer groot wordt, nadert de rekensnelheid asymptotisch tot de waarde pR(A,1), dat wil zeggen, dat het proces op den duur p keer zo snel klaar is als het op één processor uitgevoerde proces.

Geheel in lijn met de halve vectorsnelheidslengte n (paragraaf 6.2) constateren we dat precies de helft van de maximale parallelle snelheid gerealiseerd wordt als $Nk_d = pR(A,1)t_{s,p}$. De waarde van N waarvoor dat gebeurt noteren we met $N_{\frac{1}{2}}$ en R(A,p) kan hiermee herschreven worden tot

$$R(A,p) \approx pR(A,1) \frac{1}{1 + \frac{N_{\frac{1}{2}}}{N}}$$

Met R(A,1) als snelheid voor één deelproces op één processor geeft $N_{\frac{1}{2}}$ dus het aantal deelprocessen aan waarvoor op p processoren een snelheid van pR(A,1)/2 gehaald wordt. Lage synchronisatiekosten $t_{s,p}$ kunnen dus meteen vertaald worden naar een lage $N_{\frac{1}{2}}$-waarde. Een soortgelijke analyse, alsmede beschouwingen naar aanleiding daarvan met betrekking tot de CRAY X-MP/2 en de DENELCOR HEP 1, is gepubliceerd door Hockney [32].

We gaan nu nog een stapje verder. Het is heel goed denkbaar dat één enkele processor een aantal opeenvolgende operaties achter elkaar efficiënter kan uitvoeren dan elke operatie apart doet vermoeden. Dit is bij scalaire processoren vaak al het geval en we hebben het effect uitgebreid beschreven voor vectorprocessoren. Omdat in steeds meer gevallen de processoren in een parallelle computer zelf weer vectorprocessoren zijn, gaan we wat dieper in op het geval dat een vectoriseerbare loop, ter verhoging van de snelheid, wordt uitgevoerd door p vectorprocessoren, simpelweg door elke processor een p-de gedeelte van de loop voor zijn rekening te laten nemen.

Ons rekenschema $\tilde{A}$ omvat nu het uitvoeren van een aantal vectoroperaties op vectoren van gelijke lengte n en het aantal flops dat daarbij is betrokken wordt aangegeven met nk_A (k_A is het aantal flops per loop doorgang).

Volgens paragraaf 6.2 volgt voor de verwerkingstijd t_1 op één vectorprocessor:

$$t_1 = \frac{k_A}{R_\infty(\tilde{A})}(n + n_{\frac{1}{2}})\ \mu\text{sec.},$$

en voor de rekensnelheid van één processor, weergegeven door $R(\tilde{A},n,1)$, geldt

$$R(\tilde{A},n,1) = R_\infty(\tilde{A})\frac{1}{\frac{n_{\frac{1}{2}}}{n} + 1}.$$

Even voor het gemak aannemend dat n een veelvoud is van het aantal processoren p, kunnen we de vectorverwerking uitsmeren over p processoren door elke processor te laten werken op vectorsegmenten ter lengte n/p. De verwerkingstijd voor zo'n segment wordt beschreven door

$$\frac{k_A}{R_\infty(\tilde{A})}\left(\frac{n}{p} + n_{\frac{1}{2}}\right)$$

en als we weer de synchronisatiekosten (in μsec) in beeld brengen via $t_{s,p}$, dan volgt voor de verwerkingstijd op p processoren:

$$t_p = \frac{k_A}{R_\infty(\tilde{A})}\left(\frac{n}{p} + n_{\frac{1}{2}}\right) + t_{s,p}$$

$$= \frac{1}{pR_\infty(\tilde{A})}(nk_A + pn_{\frac{1}{2}}k_A + pR_\infty(\tilde{A})t_{s,p})$$

De term $pR_\infty(\tilde{A})t_{s,p}$ laat zich weer interpreteren als het aantal flops dat parallel had kunnen worden uitgevoerd in de nu aan synchronisatie bestede tijd. De term $pn_{\frac{1}{2}}k_A$ geeft het aantal flops weer dat parallel had kunnen worden uitgevoerd in de tijd die besteed is aan het opstarten van de p parallelle vectoroperaties.

Voor het prestatievermogen $R(\tilde{A},n,p)$ van de p vectorprocessoren gezamenlijk geldt:

$$R(\tilde{A},n,p) = \frac{k_A \cdot n}{t_p} = pR_\infty(\tilde{A})\frac{n}{n + pn_{\frac{1}{2}} + \frac{pR_\infty(\tilde{A})t_{s,p}}{k_A}}$$

$$= pR_\infty(\tilde{A})\frac{1}{1 + \frac{n_{\frac{1}{2},p}}{n}},$$

met $n_{\frac{1}{2},p} = \frac{pR_\infty(\tilde{A})t_{s,p}}{k_A} + pn_{\frac{1}{2}}$.

We zien dat voor toenemende n de rekensnelheid asymptotisch nadert tot $pR_\infty(\tilde{A})$. Voor

$n = n_{\frac{1}{2},p}$ wordt precies de helft van deze maximale snelheid, namelijk $\frac{p}{2}R_\infty$ ($\tilde{A}$), gehaald, vandaar de notatie $n_{\frac{1}{2}p}$. Deze waarde heeft ongeveer dezelfde betekenis als de $N_{\frac{1}{2}}$ hierboven, echter er is een belangrijk verschil. Als de synchronisatiekosten relatief verwaarloosbaar zijn ten opzichte van t_p, dat wil zeggen $t_{s,p} \approx 0$, dan geldt dat $N_{\frac{1}{2}} \approx 0$. Echter in dat geval geldt $n_{\frac{1}{2},p} \approx pn_{\frac{1}{2}}$ en deze waarde is in het algemeen allerminst gelijk aan nul. De term $pn_{\frac{1}{2}}$ wordt geïntroduceerd doordat de p loops, elk met de lengte $\frac{n}{p}$, minder effectief behandeld worden dan één lange loop met lengte n. Dit effect duiden we aan met effectiviteitsverlies. Het is dus goed er rekening mee te houden dat bij het parallel splitsen van vector loops over p processoren, de vectorlengte op zijn minst p keer zo groot moet zijn als de $n_{\frac{1}{2}}$-waarde van één enkele processor, om een snelheid in de buurt van de halve maximale snelheid te halen.

DERTIEN

PRAKTISCHE ASPECTEN VAN PARALLEL PROGRAMMEREN

In hoofdstuk 11 hebben we reeds de bedoeling van onze behandeling van parallel rekenen uiteengezet. Het vorige hoofdstuk was gewijd aan meer theoretische beschouwingen omtrent het te verwachtingen rendement van parallel werkende processorsystemen. In dit hoofdstuk zullen we alvast ingaan op algemene programmeeraspecten en een kader scheppen voor onze beschouwingen en analyses omtrent de effecten van parallellisme in de praktijk. Het is goed om eerst nog eens samen te vatten vanuit welk gezichtspunt dat zal geschieden.

13.1 Opmerkingen vooraf

a. Het soort rekenwerk

Het merendeel van de te bespreken en te analyseren berekeningen betreft relatief eenvoudige algoritmen, zoals een inprodukt, een matrix-vectorvermenigvuldiging of het oplossen van een recursie. De hiermee opgedane ervaringen op parallelle systemen geven enig idee van de programmatische aanpak en van de te realiseren snelheidseffecten. Men moet echter terughoudendheid betrachten ten aanzien van het generaliseren van deze ervaringen tot meer complexe situaties. Een enkele keer zullen we ook gewag maken van tijdmetingen aan meer complexe berekeningen, maar dit zal slechts sporadisch gebeuren en in die gevallen dient men zich te realiseren dat het tussenopnamen betreft van codes waarvan het allerminst vaststaat dat ze tot op het bot geoptimaliseerd zijn. Het onderzoek naar de meest geschikte parallelle aanpak van diverse rekenschema's is immers pas op gang gekomen en de vermelde resultaten moeten dan ook veelal

gezien worden als momentopnamen die hooguit een glimp bieden van wat de toekomst voor ons in petto heeft.

b. De programmeertaal

We hebben voor het gezichtspunt gekozen dat parallellisme een vereiste is om de hoge rekensnelheden te halen die nodig zijn om belangrijke wetenschappelijk problemen te helpen ontrafelen. Omdat het leeuwedeel van het technisch-wetenschappelijk rekenen gebeurt met behulp van FORTRAN-programma's hebben we ons gericht op het gebruik van parallelle technieken binnen deze taal. Een groot probleem hierbij is dat het parallel programmeren binnen FORTRAN niet goed geregeld is en dat de diverse fabrikanten gezocht hebben naar uitbreidingen van de taal om parallel rekenen mogelijk te maken. We zullen trachten de verschillende varianten aan bod laten komen.

c. Het soort parallelle computer

We beschouwen uitsluitend parallelle computers waarvan de processoren asynchroon min of meer grote brokken van een programma, en zelfs hele complete programma's, onafhankelijk van elkaar kunnen uitvoeren. Dat maakt het ons mogelijk de verwerkingstijd van een berekening op p processoren te vergelijken met de rekentijd die gebruikt wordt door één processor in zijn uppie. Voor de gebruiker is het op dit soort computers mogelijk om af te wegen of het zinvol genoeg is p processoren voor één karwei in te schakelen of dat men er meer bij gebaat is wanneer een aantal processoren aan verschillende programma's werken. Dit houdt wel in dat we ons niet zullen bemoeien met parallelle systemen die bestaan uit een (zeer groot) aantal relatief zeer eenvoudige processortjes die allemaal tegelijk eenzelfde basisopdracht kunnen uitvoeren. Machines als de ICL-DAP en Connection Machine, hoe interessant ook, blijven hierdoor onbesproken.

d. De machines in concreto

Verder beperken we ons tot redelijk bekende, commercieel verkrijgbare machines en laten we de vele interessante machines die in de onderzoeksfeer worden ontwikkeld (zoals bijvoorbeeld het CEDAR-project in Illinois, SUPREMUM in Bonn en de DPP van de TU Delft) buiten beschouwing. De machines moeten verder in staat geacht worden een redelijk breed assortiment van technisch wetenschappelijke toepassingen te lijf te kunnen en machines die hoofdzakelijk ontworpen zijn om ingezet te kunnen worden voor een beperkte klasse problemen (bijvoorbeeld Fast Fourier Transforms) worden hier ook niet besproken. Bovendien vonden we het gewenst zelf enige ervaring met de besproken systemen te hebben en dat houdt natuurlijk ook een behoorlijke beperking in. Tenslotte moesten de systemen ook redelijk in FORTRAN aanspreekbaar zijn, waardoor transputersystemen als die van Meiko in eerste instantie afvielen.

Al deze beperkingen hebben ertoe geleid dat de te bespreken parallelle systemen slechts over een bescheiden aantal parallelle processoren beschikken, zeg 16 of minder. Gezien de stand van de techiek, ten aanzien van verbindingsproblemen en de daaruit voortvloeiende synchronisatie kosten lijkt dat, gezien onze voorbeschouwingen in hoofdstuk 12, ook al heel mooi en moeten we eerst maar eens zien hoe we met een bescheiden aantal processoren het onderste uit de kan kunnen halen. Op grond van onze beschouwingen en ervaringen kunnen we evenwel zekere verwachtingen koesteren ten aanzien van het gebruik van meerdere processoren.

13.2 Parallellisme in FORTRAN-programma's

Met de door ons beschouwde parallelle computers kunnen we grofweg drie verschillende niveaus van parallelle verwerking realiseren:

1. parallellisme op job niveau: de CPU's verwerken elk een gehele job of zijn samen bezig aan grotere eenheden van een job (zoals bijvoorbeeld gelijktijdige compilatie van onderdelen van een programmapakket);
2. parallellisme op subroutine niveau: de CPU's verwerken parallel grotere delen van een programma (bijvoorbeeld gelijktijdige uitvoering van subroutine aanroepen);
3. parallellisme op DO-loop niveau: de doorgangen van een DO-loop worden parallel verwerkt (of in vectormode of in beide).

Uiteraard zijn ook combinaties mogelijk en kunnen bijvoorbeeld grote brokken van een algoritme parallel uitgevoerd worden waarbij binnen elke brok weer vectorisatie kan worden toegepast. Wij richten ons verder op de onder 2 en 3 genoemde vormen van parallellisme.

De programmeertechnieken voor shared memory systemen en message passing systemen zijn essentieel verschillend. Belangrijke verschilpunten zijn:

a. shared memory systemen

1. in een programma geeft men expliciet aan welke onderdelen parallel verwerkt kunnen worden;
2. men moet zelf synchronisatiepunten inbouwen (waardoor processen op elkaar moeten wachten) om ervoor te zorgen dat de CPU's de datalocaties in het gemeenschappelijk geheugen in de goede volgorde aanspreken en niet bijvoorbeeld data overschrijven die nog door andere CPU's gebruikt hadden moeten worden;
3. parallellisme kan benut worden zowel op niveau 2 als 3.

b. message passing systemen

1. de processoren worden elk afzonderlijk geprogrammeerd. Elke processor heeft zijn eigen lokale geheugen en data-uitwisseling (communicatie) vindt plaats door

het expliciet verzenden van data en het wachten met verdere uitvoering tot de gewenste data gearriveerd zijn;

2. de processoren worden ingezet voor parallellisme in een programma op niveau 2. Wanneer de processoren zelf weer vectorprocessoren zijn dan komt per processor ook niveau 3 aan de orde.

Net zoals bij het programmeren van vectormachines heeft elke fabrikant specifieke uitbreidingen op de taal FORTRAN verzonnen om parallellisme in een programma aan te geven. Voor vectormachines is er inmiddels enige duidelijkheid ontstaan omtrent de faciliteiten die FORTRAN zou moeten bieden en de vernieuwde taal FORTRAN 8X zal dan ook vectorstatements omvatten (die grote gelijkenis vertonen met de CDC CYBER 205 vectorsyntax, maar daar zal Control Data wel op geanticipeerd hebben). Mede op grond van de grote onderlinge verschillen tussen parallelle systemen is het voor wat betreft parallelle uitdrukkingsmechanismen nog lang zo ver niet. Reid [49] merkt op, in een tussentijdse evaluatie van de FORTRAN 8X commissie, dat er momenteel nog te weinig overeenstemming is om tot een bruikbare standaardisatie te komen. Hij geeft nog wel aan dat er al reeds in FORTRAN 77 een (onvoorziene?) mogelijkheid is om parallellisme tot uitdrukking te brengen door gebruik te maken van de eis dat het resultaat van een function niet mag afhangen van de volgorde waarin de argumenten geëvalueerd worden. Door de argumenten mee te geven in de vorm van expressies of function-calls kan men op deze wijze (impliciet) aangeven dat deze argumenten parallel geëvalueerd mogen worden. Een eenvoudig voorbeeld hiervan is het splitsen van de sommatie van een rij getallen volgens

```
SUM = SUM1(SUM2(A,N/2),SUM3(A(N/2 + 1),N - N/2)) .
```

Misschien heeft Reid met dit voorbeeld zijn gevoel voor Britse understatement laten meespreken, want het lijkt ons een wat omslachtige methode voor het parallelliseren van wat grotere programma's.

Op het eerste gezicht zou het kunnen lijken dat de message passing systemen de hoogste eisen aan de taal stellen, maar dat is niet zo. Dit soort systemen kan meestal in standaard FORTRAN, zonder extra taalconstructies, worden toegesproken. Het enige dat men in principe nodig heeft zijn speciale routines om daarmee het datatransport tussen de processoren te bewerkstelligen.

Voor het efficient gebruik van shared memory systemen zijn verschillende hulpmiddelen bedacht, die er globaal op neerkomen dat de programmeur in een programma kan aangeven dat

- DO-loops parallel verwerkt kunnen worden. Dit kan door het gebruik van een afwijkende DO-constructie; echter meestal kan het worden aangegeven door een speciale comment directive voor de loop te plaatsen (zoals bij het afgedwongen vectoriseren);
- subroutines of grotere programmaonderdelen (processen) parallel verwerkt kunnen worden. Dat gebeurt bijvoorbeeld weer via speciale comment directives maar het kan soms ook gerealiseerd worden door aanroep van speciale subroutines waarmee men parallelle subtaken kan creëren. Voor synchronisatiedoeleinden

worden soms speciale variabelen gedefinieerd (bijvoorbeeld beginnend met een $-teken), die een soort vlagfunctie hebben (een actie heeft al dan niet plaats gevonden). Soms worden ook speciale subroutines gebruikt voor synchronisatie en soms worden de 'vlagvariabelen' (semaforen) in een gereserveerd COMMON blok gezet.

Het is natuurlijk gewenst dat men de inspanning, die verricht moet worden om een groot programma geschikt te maken voor parallelle verwerking, zoveel mogelijk kan beperken en dat bovendien de daarvoor benodigde mechanismen niet meteen onder je handen verouderen. Verder is het van groot belang dat er een hoge mate van overdraagbaarheid naar andere machines gewaarborgd blijft. Om die redenen is het niet aantrekkelijk om veelvuldig gebruik te moeten maken van niet-standaard FORTRAN-constructies. Vanwege de al met al toch grote programmeerarbeid die voor het herstructureren van grote produktiecodes verricht moet worden, is het verder verstandig om gebruik van die constructies te vermijden die op de nominatie staan om te zijner tijd uit de taal FORTRAN te verdwijnen (zo zou de COMMON-constructie volgens plan op de nominatie staan om na FORTRAN 9Y te verdwijnen [49]).

De verschillende parallelle programmeertechnieken komen redelijk aan bod bij de computersystemen die wat nader bekeken zullen worden: SEQUENT Balance 8000, ALLIANT FX/8, CRAY X-MP/4 (shared memory systemen) en de NCUBE/4 (message passing systeem). Voor zover bekend vertonen de programmeerhulpmiddelen voor deze systemen grote overeenkomst met wat er verder zoal op de markt is. Een goed overzicht van de verschillende (geabstraheerde) technieken wordt gegeven door Karp [38].

Om het effect van parallellisme te bestuderen zullen we vaak gebruik maken van een modelprobleem. We hebben daarvoor gezocht naar een algoritme dat eenvoudig genoeg was om niet te verzinken in algoritmische details, maar ook ingewikkeld genoeg om diverse parallelle aspecten te kunnen demonstreren. Het uitsplitsen van de sommatie van een rij getallen in een aantal deelsommaties leek wat al te flauw en voor de hand liggend. In de meeste gevallen zal men liefst grotere eenheden in een rekenschema parallel wensen te verwerken om synchronisatieëffecten zoveel mogelijk te beperken. De volle matrixvermenigvuldiging is ook niet zo geschikt omdat hier teveel vanzelfsprekend parallellisme aan vastzit. Alle inprodukten tussen de matrixrijen en de vector kunnen immers onafhankelijk van elkaar worden uitgevoerd. Een dergelijke eenvoudige vorm van parallellisme zal men in de praktijk slechts zelden tegenkomen.

Veel realistischer en interessanter is een situatie waarin een serieel algoritme vervangen moet worden door een parallel alternatief (dat serieel uitgevoerd duurder geweest zou zijn dan het oorspronkelijke seriële algoritme zelf). Als die parallelle variant dan nog wat keuzevrijheden ten aanzien van parallellisme toelaat, tussentijdse synchronisatiepunten vereist en wellicht een stukje seriële code omvat dan hebben we een interessante mix. Een relatief eenvoudig probleem waarbij al deze aspecten een rol spelen is het reeds eerder besproken verdeel-en heersalgoritme voor het oplossen van (grote) bidiagonale stelsels. Door dit algoritme als voorbeeld te nemen willen we overigens niet suggereren dat het voor alle besproken computersystemen de optimale keuze zou zijn.

13.3 Parallelle aspecten van het verdeel- en heersalgoritme

Het algoritme zelf is reeds uitvoerig besproken in paragraaf 8.3.c in het kader van vectorisatietechnieken. We gaan van die bespreking uit en lichten de parallelle aspecten toe.

In de eerste fase van het proces worden er m recursies, elk ter lengte k, uitgevoerd in plaats van de oorspronkelijke recursie die een lengte n = mk had. Deze fase is uiteraard zeer parallel en dit parallellisme wordt in een FORTRAN programma het duidelijkst naar voren gebracht door de berekening van de elementen b(i,j) en a(i,j) (met j vast, j geeft het nummer van de subgroep aan) bij elkaar te houden in de binnenste loop (in tegenstelling tot de situatie bij het vectoriseren).

De berekeningen in fase 1 voor de subgroepen kunnen parallel verwerkt worden (dat wil zeggen dat een processor steeds één gehele subgroep behandelt) met behulp van uitsluitend de data die bij de betreffende subgroep horen. Dat is prettig voor de message passing systemen waarbij elke processor een eigen lokaal geheugen heeft. Elk van de parallel uit te voeren berekeningen bestaat zelf uit een recursie ter lengte k. Deze berekeningen zijn dus minder geschikt voor vectorverwerking. Wanneer de processoren zelf weer vectorprocessoren zijn dan zou men er beter aan doen om uit te gaan van de vectorcode uit paragraaf 8.3.c en vervolgens de daarin optredende j-loop over de processoren te verdelen (aannemend dat m groot genoeg is om dat zinvol te doen zijn).

De behandeling van de correctiefase in het algoritme is wat minder triviaal. In principe kunnen alle elementen van de j-de groep gecorrigeerd worden zodra het laatste element van de (j - 1)-ste groep bekend is. Dat betekent dat de berekening voor de j-de groep (een loop met lengte k) uitgesmeerd moet worden over de overige processoren en vergeleken met de eerste fase van het proces heeft elke processor nu met veel kleinere brokjes werk te maken hetgeen deze variant nogal gevoelig maakt voor synchronisatiekosten. Deze oplossing heeft voor message passing systemen bovendien een nog veel groter nadeel. De correctiecoëfficienten voor de j-de groep staan hier na afloop van de eerste fase in het lokale geheugen van de processor die de j-de groep behandeld heeft. Voor parallelle correctie van de j-de groep moeten deze data eerst expliciet verzonden worden naar de overige processoren. Dat leidt tot flink wat transportkosten. We zullen bij de bespreking van de NCUBE/4 nog uitgebreid op dit probleem terugkomen.

Volgens de zogenaamde aanpak II van de tweede fase (zie paragraaf 8.3.c) worden eerst de laatste elementen van elke groep gecorrigeerd, waarna de correctie van alle overige elementen weer onafhankelijk, dat wil zeggen: parallel, kan geschieden. Dus ten koste van een seriële stap voor de berekening van de laatste elementen van elke groep bereiken we zo dat daarna de parallelle verwerking kan plaatsvinden met grotere brokken en we bereiken dat de correctie van de overige elementen van groep j ook weer kan gebeuren door de processor die de correctiecoëfficiënten in het lokale geheugen heeft (van belang voor message passing systemen). In het laatste geval hoeft de j-de processor slechts de waarde van het laatste element van de (j - 1)-ste groep te ontvangen.

Indien we weer toestaan dat het rechterlid B overschreven mag worden door de uiteindelijke oplossing en dat de recursiecoëfficiënten A overschreven mogen worden door de correctiecoëfficiënten, dan komt het hele algoritme inclusief fase 1 en aanpak II voor fase 2 er als volgt uit te zien:

```
C     FASE 1 VAN HET VERDEEL- EN HEERS ALGORITME
        DO 20 J = 1,M
          DO 10 I = 2,K                                    } parallel
            B(I,J) = B(I,J) + A(I,J) * B(I-1,J)            } te
            IF (J.NE.1) A(I,J) = A(I,J) * A(I-1,J)         } verwerken
   10   CONTINUE
   20   CONTINUE
C     FASE 2 VAN HET PROCES VOLGENS AANPAK II,
C     EERST WORDT DE CORRECTIE VAN DE LAATSTE ELEMENTEN
C     UITGEVOERD:
        DO 30 J = 2,M
          B(K,J) = B(K,J) + A(K,J) * B(K,J-1)              } serieel proces
   30   CONTINUE
C     DAARNA VOLGT DE CORRECTIE VAN DE OVERIGE ELEMENTEN
        DO 50 J = 2,M
          DO 40 I = 1,K-1                                  } parallel
            B(I,J) = B(I,J) + A(I,J) * B(K,J-1)            } te
   40     CONTINUE                                         } verwerken
   50   CONTINUE
```

We wijzen er nogmaals op dat de keuze op dit algoritme is gevallen om een aantal aspecten van het parallel rekenen te kunnen demonstreren. Voor de lezer, die zich afvraagt hoe het algoritme op zich te vergelijken is met het originele seriële recursieproces, merken we op dat het verdeel-en heers algoritme ongeveer 2.5 keer zo veel flops vergt. Men zal dus op een parallel systeem minstens een factor 2.5 ten gevolge van parallellisme moeten verdienen om althans het seriële proces op één processor te verslaan. In [59] is aangetoond dat het verdeel- en heersalgoritme voor de meest voorkomende recursies doorgaans vrijwel net zo nauwkeurig werkt ten aanzien van afrondfouten als het seriële recursieproces.

VEERTIEN

SHARED MEMORY SYSTEMEN

14.1 Parallelle scalaire processoren

Het voor de beginnende parallellist meest doorzichtige systeem verkrijgt men als een aantal scalaire processoren gekoppeld worden aan een gemeenschappelijk geheugen. De technische realisatie van een efficiënt werkend systeem is niet zo eenvoudig, vaak zit er nog een cache geheugen tussen processor en geheugen om te voorkomen dat alle processoren tegelijk het geheugen aanspreken. Een gedeelte van de data, alsmede lokale variabelen, worden in cache opgeslagen en er wordt steedt gecontroleerd of benodigde data aldaar aanwezig zijn. Het principe doet denken aan het page-mechanisme bij virtual memory (zie hoofdstuk 10). Wij zullen ons niet bezighouden met dit soort details omdat de programmeur meestal niet in staat wordt gesteld rechtstreeks van het cache geheugen gebruik te maken en bij de programmering kan doen alsof alles zich in een centraal geheugen bevindt. Een dergelijk soort systeem wordt gerealiseerd in de SEQUENT Balance computers.

Wij hebben een paar experimenten uitgevoerd op een SEQUENT Balance 8000 met 12 scalaire processoren. SEQUENT brengt inmiddels verbeterde, snellere en uitgebreidere modellen op de markt, echter het principe van parallelle scalaire processoren met een gemeenschappelijk geheugen laat zich heel goed illustreren met de door ons medio 1986 opgedane ervaringen.

Bij deze computers kan men naar keuze over alle of een deel van de processoren beschikken. Het is dus denkbaar dat twaalf gebruikers elk één processor bezetten en het kan evenzeer voorkomen dat één gebruiker vijf processoren reserveert voor een programma, daarbij de overige zeven processoren vrijlatend voor andere gebruikers.

Het moge duidelijk zijn dat men met een gering aantal scalaire processoren de grenzen van de wetenschap niet drastisch verlegt, maar daar is de machine ook niet voor

bedoeld. Bij SEQUENT denkt men aan succesvolle toepassing in typische database- en gegevensverwerkingssituaties, zoals kantoorautomatisering, voorraadbeheer, patiëntenadministraties, hotelreserveringsystemen, en dergelijke.

Voor ons doet op dit moment de snelheid van het systeem zelf niet zozeer ter zake, al moet men er wel rekening mee houden dat allerlei synchronisatie kosten relatief minder zwaar wegen naarmate de floating point processoren trager zijn.

Het belangrijkste hulpmiddel voor het aangeven van parallellisme in een FORTRAN-programma op de SEQUENT is de C$DOACROSS directive. Deze wordt onmiddellijk voorafgaand aan de parallel uitvoerbare DO-loop geplaatst. Wat er verder binnen de DO-loop gebeurt doet niet zoveel ter zake, dat wil zeggen: de hoeveelheid werk mag groot zijn en er mogen subroutine-aanroepen in plaatsvinden. Een mogelijkheid om de uitvoering van drie verschillende subroutines A, B en C parallel te doen plaatsvinden is bijvoorbeeld:

```
      DO 10 I = 1,3
        IF (I.EQ.1) CALL A (   )
        IF (I.EQ.2) CALL B (   )
        IF (I.EQ.3) CALL C (   )
10    CONTINUE
```

Met de C$DOACROSS directive geeft men aan welke variabelen slechts lokaal betekenis hebben (cache), zoals bijvoorbeeld loop parameters en welke waarden ook buiten de loop betekenis hebben. Verder kan men nog aangeven dat niet elke doorgang apart door een processor verwerkt moet worden, maar dat verwerking plaats moet vinden op groepjes doorgangen (loop unrolling). Het is zaak de groepsgrootte, aangegeven met de parameter CHUNK, niet te klein te kiezen om synchronisatiekosten relatief te reduceren, maar ook weer niet te groot. Naarmate er namelijk meer groepjes te verdelen zijn over de processoren, ziet het besturingssysteem beter kans zorg te dragen voor een evenwichtige verdeling van het werk over alle processoren. Dit speelt een belangrijke rol wanneer niet elke loop doorgang evenveel werk kost. In onze testgevallen hebben we de loop steeds in groepjes van vijf doorgangen onderverdeeld. Het relevante deel van het programma komt er dan als volgt uit te zien:

```
C$DOACROSS, LOCAL (I,J), SHARED (M,K,A,B), CHUNK = 5
     DO 20 J = 1,M                                    } de doorgangen
       DO 10 I = 2,K                                  } van de DO-loop
         B(I,J) = B(I,J)+A(I,J)*B(I-1,J)              } worden parallel
         IF (J.NE.1) A(I,J)=A(I,J)*A(I-1,J)           } uitgevoerd in
 10    CONTINUE                                        groepjes van 5
 20  CONTINUE
     DO 30 J = 2,M                                    } wordt serieel
       B(K,J) = B(K,J)+A(K,J)*B(K,J-1)                } uitgevoerd
 30  CONTINUE
```

```
C$DOACROSS, LOCAL (I,J), SHARED (M,K,A,B), CHUNK = 5
      DO 50 J = 2,M
        DO 40 I = 1,K-1
          B(I,J) = B(I,J)+A(I,J)*B(K,J-1)
   40   CONTINUE
   50 CONTINUE
```

} de doorgangen van de DO 50-loop worden weer parallel uitgevoerd in groepjes van 5

Voor onze experimenten bleek gemiddeld CHUNK = 5 de beste resultaten te leveren, de effecten waren overigens marginaal: CHUNK = 1 leverde nagenoeg dezelfde rekentijden op.

Voordat we wat laten zien van de behaalde resultaten zullen we eerst eens proberen na te gaan wat we zoal aan versnellingseffecten zouden mogen verwachten. In de analyse geven we weer met $\lceil a \rceil$ afronding van a naar boven aan.

De DO 20-loop wordt volgens de directive gesplitst in $\lceil \frac{m}{5} \rceil$ groepjes. Dit aantal wordt verspreid over p processoren zodat er processoren zijn die $\lceil \lceil \frac{m}{5} \rceil / p \rceil$ groepjes moeten verwerken. Deze processoren zijn bepalend voor de rekentijd van het systeem.

Als we de rekentijd voor de 3 flops uit de binnenste loop op 3α seconde stellen, dan is de 'meest bezige' processor (in het geval dat het aantal groepjes geen veelvoud is van p, zijn sommige processoren zwaarder belast dan andere) gedurende een tijd

$$\lceil \lceil \tfrac{m}{5} \rceil / p \rceil \cdot 5 \cdot (k-1) \cdot 3\alpha \text{ sec.}$$

aan het werk. De recursie aan het begin van de tweede fase vergt dan 2(m-1) sec. en de parallelle verwerking van de DO 50-loop vergt

$$\lceil \lceil \tfrac{m-1}{5} \rceil / p \rceil \cdot 5 \cdot (k-1) \cdot 2\alpha \text{ sec.}$$

De totale rekentijd van het gehele systeem bedraagt dan

$$t_p = \left\{ (3 \lceil \lceil \tfrac{m}{5} \rceil / p \rceil + 2 \lceil \lceil \tfrac{m-1}{5} \rceil / p \rceil) 5(k-1) + 2(m-1) \right\} \alpha + t_{s,p} \text{ sec.}$$

We hebben de term $t_{s,p}$ toegevoegd om de synchronisatiekosten bij gebruik van p processoren in rekening te brengen. De waarde van $t_{s,p}$ hangt (zwak) van de CHUNK-parameter af.

De speed-up (of parallelle versnellingsfactor) S_p wordt gedefinieerd als

$$S_p = \frac{t_1}{t_p} \text{, waarbij } t_{s,1} = 0 .$$

Wanneer we uitgaan van een ideale situatie zonder synchronisatiekosten, dus $t_{s,p} = 0$, dan zou voor k = 100, m = 360 en p = 8 de versnellingsfactor de volgende waarde moeten hebben

$$S_8 = \frac{(3*360 + 2*359) * 99 + 2*359}{(3\lceil \lceil \frac{360}{5} \rceil / 8 \rceil + 2 \lceil \lceil \frac{359}{5} \rceil / 8 \rceil) * 5 * 99 + 2*359} \approx 7.78$$

De werkelijk gemeten tijden bedroegen $\tilde{T}_1 = 7.63$ sec. en $\tilde{T}_8 = 1.13$ sec. en de gemeten versnellingsfactor $\tilde{S}_8$ bedraagt dus $\tilde{S}_8 = 6.75$.

Op grond van de theoretische waarde van S_8 zou, uitgaande van $\tilde{T}_1$ de waarde van $\tilde{T}_8$ ongeveer $\tilde{T}_1/7.78 = 0.978$ hebben moeten zijn. Hieruit en uit de werkelijk gemeten waarde voor $\tilde{T}_8$ concluderen we dat de synchronisatiekosten ongeveer gelijk zijn aan $t_{s,8} \approx 0.15$ sec. Uit soortgelijke overwegingen volgde met behulp van de gemeten tijden voor deze vaste waarden van m en k voor $t_{s,p}$:

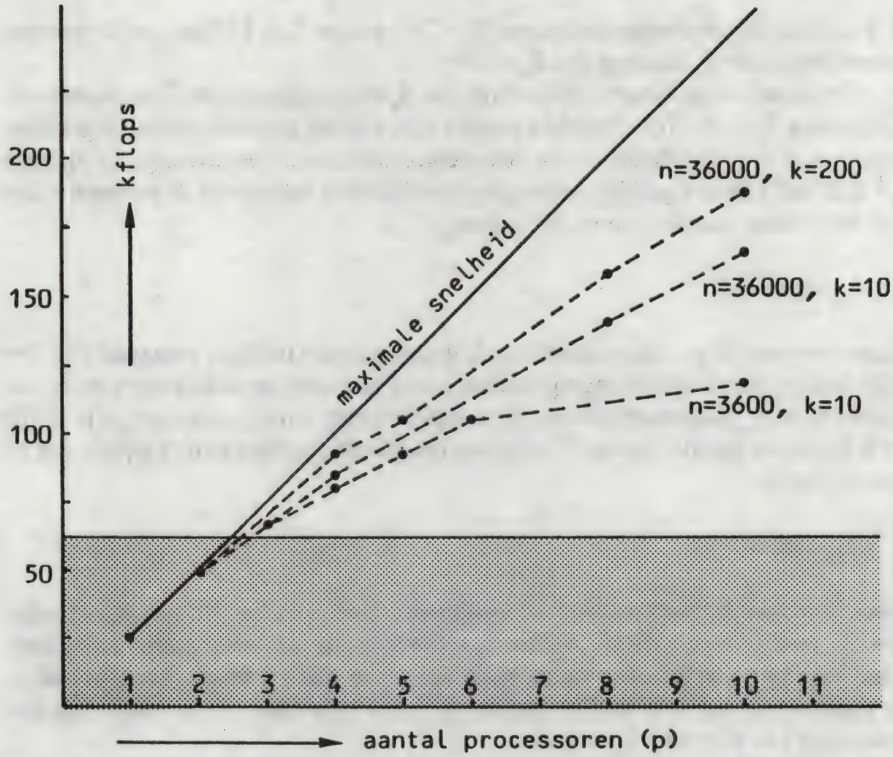
$$t_{s,p} \approx p * 0.02.$$

Wanneer we met de gemeten waarde van $\tilde{S}_8$ te rade gaan in tabel 2 uit paragraaf 12.2 dan blijkt dat een drastische verhoging van het aantal processoren weinig extra meer zou opleveren. Als $t_{s,p}$ inderdaad ook voor grotere p evenredig is met p volgens $t_{s,p} \approx p * 0.02$, dan schatten we met behulp van $\tilde{T}_1$ voor een denkbeeldig gelijksoortig systeem van 36 processoren dat

$$\tilde{T}_{36} \approx 0.961$$

Deze laatste waarde houdt een versnellingsfactor in van $\tilde{S}_{36} \approx 7.9$ en we zien dat het onder deze omstandigheden geen zin heeft met veel meer dan acht processoren te werken. Indien men kans zou zien de synchronisatiekosten relatief verwaarloosbaar te maken dan zouden we, volgens tabel 2 (paragraaf 12.3) een versnellingsfactor hebben mogen verwachten van ongeveer $S_{36} = 31.6$.

In figuur 12 geven we de gemeten snelheid van de SEQUENT Balance 8000 aan voor diverse gevallen als functie van het aantal processoren. Met de gestippelde lijn is aangegeven p keer de snelheid van 1 processor, ofwel de asymptotische snelheid van het p-processorsysteem (voor voldoend grote waarden van n en k). Als men tenslotte het algoritme nog wenst te vergelijken met het seriële proces op één processor dan ziet men dat de maximale winstfactor bij p processoren iets in de orde van 2p/5 bedraagt.



Figuur 12. Rekensnelheid van de SEQUENT Balance 8000 voor het verdeel-en-heers-algoritme.

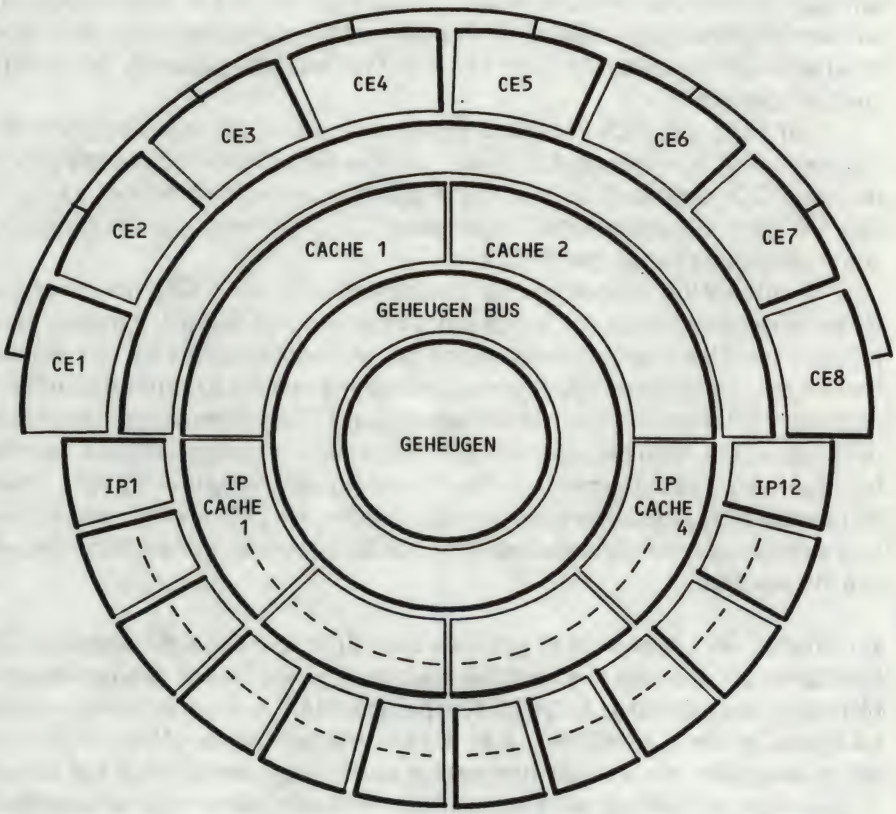
Het gearceerde gebied geeft aan onder welke omstandigheden men sneller uit is door de oorspronkelijk recursie op slechts één processor te laten verwerken.

14.2 Parallelle vectorprocessoren

Een machine die opgebouwd is uit een aantal parallel te gebruiken vectorprocessoren is de volgende logische stap op weg naar hogere rekensnelheden. We moeten zo'n systeem vooral niet verwarren met het verschijnsel van de meervoudige vector pipe zoals dat werd toegepast bij de CYBER 205 en de Japanse supercomputers. Bij een meervoudige vector pipe worden slechts bepaalde functional units, zoals bijvoorbeeld de optteller, binnen een processor meervoudig uitgevoerd zodat een vectoroperatie kan worden opgesplitst in een overeenkomstig aantal vectorstromen. Die vector pipes kunnen dus alleen één lange vectoroperatie opsplitsen, zij zijn niet geschikt voor het parallel uitvoeren van geheel verschillende vectoropdrachten.

Bij een parallel vectorprocessorsysteem worden in feite een paar complete vectorcomputers aan één gemeenschappelijk geheugen gekoppeld. Door gebruik te maken van goedkopere (= tragere) componenten en een eenvoudig koelingssysteem verliest men dan in zuivere rekensnelheid per processor, maar hoopt men de winst te behalen door gebruik te maken van parallellisme. Het is gebleken dat men op die manier een voor veel wetenschappelijke toepassingen aantrekkelijk systeem kan construeren.

De ALLIANT FX/8 bestaat uit één tot acht relatief trage vectorprocessoren die via een cache (groepsgewijs) op een centraal geheugen zijn aangesloten (zie figuur 13).



Figuur 13

In principe kan de machine uit minimaal één en maximaal acht vectorprocessoren (CE's = Computational Elements) bestaan. Een niet-maximale configuratie kan alsnog worden uitgebreid door er losse CE's aan toe te voegen.

Een ALLIANT FX/8 kan voor de verwerking van programma's geherconfigureerd worden in afzonderlijke complexen van CE's. Elke CE kan apart door een gebruiker benut worden en in zo'n geval hebben we in feite acht aparte (vector)computers ter beschikking. Een enkele CE gedraagt zich qua werking zo ongeveer als een CRAY-1 of een CONVEX C-1, zij het dat de klokcyclus langer is, namelijk 170 ns tegen 12.5 ns voor de CRAY-1 en 100 ns voor de CONVEX C-1. We gaan er bij deze beschouwingen over snelheid steeds van uit dat de berekeningen worden uitgevoerd in ongeveer 64 bits floating point (dat is CRAY en Control Data standaard precisie). Voor de ALLIANT betekent dat rekenen met DOUBLE PRECISION variabelen (de machine staat ook rekenen in (IBM) single precision (≈32 bits) toe).

De maximale rekensnelheid van een CE zou in de buurt van 11.7 Mflops moeten liggen (namelijk op grond van 170 ns en chaining van + en *, zie hoofdstuk 5); in de praktijk lijken de grootte van de vectorregisters en het transport tussen cache en centraal geheugen een limiterende factor te zijn. Een snelheid van rond de zeven Mflops voor relevante rekenproblemen schijnt wel zo ongeveer het best haalbare te zijn (zie bijvoorbeeld snelheidsmetingen in [42]. Op de invloed van het cache geheugen zullen we nog apart terugkomen.

Een ALLIANT FX/8 kan ook in logische complexen van ongelijke grootte worden opgesplitst, bijvoorbeeld 4, 2, 1 en 1 CE. Een gebruiker die op een complex van meerdere CE's werkt heeft dan ook voor de gehele duur van de job de beschikking over deze CE's. Het kan, gezien tabel 2 in paragraaf 12.2 zeker verstandig zijn soms genoegen te nemen met minder dan 8 CE's.

Waarschijnlijk zal het gebruik van bijvoorbeeld een 4 CE-complex per uur rekentijd wel duurder zijn dan het gebruik van een enkele CE en het ligt mede aan dit prijsverschil of het verkregen parallellisme daar tegen opweegt. Om een concreet voorbeeld te geven: volgens recente gegevens kost een beginconfiguratie met slechts één CE ongeveer 250 000 gulden. Een uitbreiding naar een 4 CE-systeem zou nog eens hetzelfde bedrag vergen. Wint men voor bepaalde toepassingen meer dan een factor 2 aan snelheid dan is het voor die toepassingen dus lonend om uitbreiding te overwegen. Er kunnen uiteraard ook andere overwegingen een rol spelen, een 2 CE-machine heeft een twee keer zo hoge capaciteit als een enkele CE, maar dat is voor onze parallelle beschouwingen niet relevant.

De filosofie van Alliant is dat de gebruiker niet zelf de werklast expliciet over de CE's hoeft te verdelen en zich niet nodeloos hoeft bezig te houden met synchronisatieproblematiek. Net zoals bij de SEQUENT, in paragraaf 14.1, verdeelt het systeem zelf het rekenwerk op een zo gunstig mogelijke wijze over de processoren. Voor wat dit aspect betreft vergelijken we het maar even met het meervoudige pipe principe van sommige vectorcomputers, ook daar vindt automatische verdeling van de vectorstromen plaats.

Voor parallelle verwerking op een Alliant komen alleen DO-loops in aanmerking. We kunnen voor een enkelvoudige DO-loop vier niveaus van verwerking onderscheiden:

1. scalaire verwerking : de loop wordt serieel uitgevoerd op één CE;
2. vectorverwerking : de loop wordt uitgevoerd met vectoroperaties op één CE;
3. parallelle scalaire verwerkingen : de loop doorgangen worden over de CE's verdeeld en elke loop doorgang wordt serieel verwerkt (dezelfde situatie als in paragraaf 14.1);
4. parallelle vectorverwerking : de loop wordt opgedeeld in een aantal vectoroperaties over de CE's (vergelijk dit met de vector pipes op bijvoorbeeld de CDC CYBER 205).

Net zoals bij de SEQUENT wordt de gebruiker, ten koste van de beperking tot DO-loops, de ellende van het synchroniseren van grotere acties bespaard (zie paragraaf 14.1). We lichten de basisvormen van verwerking van een DO-loop toe aan de hand van een voorbeeld, ontleend aan [29] (een aardige publicatie die veel wetenswaardigs over parallel rekenen bevat). De vermelde rekentijden zijn overgenomen uit [29].

We bekijken de mogelijke verwerkingsvormen van:

```
DO 10 I = 1, 256
A(I) = A(I) + P
10   CONTINUE
```

1. *Scalaire verwerking (op 1 CE)*

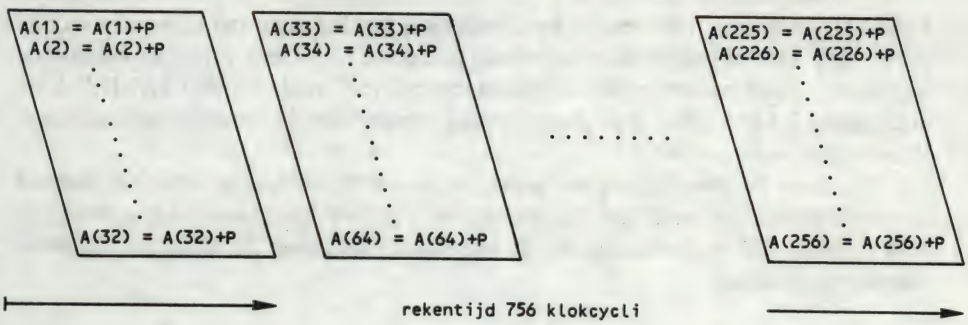
   Bij (geforceerde) scalaire verwerking worden alle opdrachten serieel verwerkt op 1 CE:

   ; A(1) = A(1) + P ; A(2) = A(2) + P ; . . . . . ; A(256) = A(256) + P;

   ——————→ rekentijd 3827 klokcycli ——————→

2. *Vectorverwerking (op 1 CE)*

   Men kan forceren dat de loop in vectormode wordt uitgevoerd op 1 CE. Wegens een vectorregisterlengte van 32 wordt de loop vectorieel in porties van 32 op 1 CE uitgevoerd:



3. *Parallelle scalaire verwerking op 8 CE's*

Men kan forceren dat de loop wel parallel maar niet vectorieel wordt verwerkt op 8 CE's. We krijgen dan schematisch het volgende beeld:

CE0: A(1) = A(1) + P; A(9) = A(9) + P; ..... ; A(249) = A(249) + P;
CE1: A(2) = A(2) + P; A(10) = A(10) + P; ..... ; A(250) = A(250) + P;
. . . .
. . . .
CE7: A(8) = A(8) + P; A(16) = A(16) + P; ..... ; A(256) = A(256) + P;

——————→ rekentijd 499 klokcycli ——————→

4. *Parallelle vectorverwerking op 8 CE's*

Onder optimale omstandigheden kan de loop in parallelle parten over de CE's verdeeld worden en wordt elk part vectorieel door een CE verwerkt (in feite wordt de verwerking onder 3 gevectoriseerd) (zie de figuur hiernaast):

Merk op dat het bij kortere vectorlengten, bijvoorbeeld n = 8, aantrekkelijker kan zijn om methode 3 te gebruiken in plaats van methode 2 of 4, vanwege de opstarttijden in de vectorpipes.

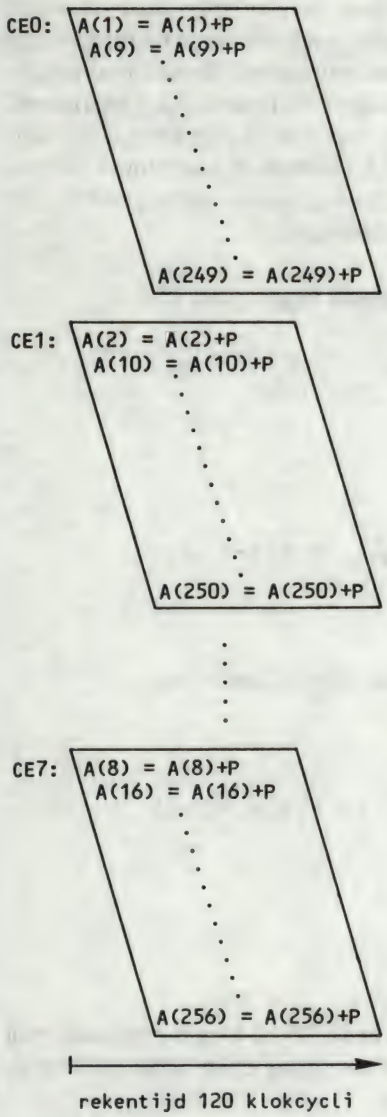
Bovendien kent de FORTRAN vertaler (van Alliant) nog de mogelijkheid om bij geneste loops de buitenste loop parallel uit te voeren en eventuele binnenste loops te vectoriseren. Dit wordt aangeduid met de term COVI (= Concurrent Outer Vector Inner). De matrixvectorvermenigvuldiging is een handeling die daarvan kan profiteren:

```
      DO 20 I = 1,N
         A(I) = 0.
         DO 10 J = 1,N                    (inprodukt)
            A(I) = A(I) + B(I,J) * C(J)
10       CONTINUE
20    CONTINUE
```

De DO-20 loop kan parallel over de CE's worden uitgesmeerd en per CE kan het binnengedeelte (de inproduktberekening) weer vectorieel worden uitgevoerd (zie voor vectorverwerking van het inprodukt bijvoorbeeld paragraaf 7.2). Deze wijze van berekenen is voor de Alliant gunstiger dan de 'kolomaccumulatie'-methode (met SAXPY's) die in paragraaf 7.3 besproken is en die zo gunstig uitpakt voor de meeste vectormachines. Voor tijdmetingen zie [42].

Wanneer de compiler zelf niet in staat is een vectoriseerbare situatie te herkennen (wegens vermeende data afhankelijkheden), dan kan vectorisatie worden afgedwongen door onmiddellijk voorafgaand aan de betreffende DO-loop de volgende comment directive te plaatsen:

(bij langere loop lengte vindt hier weer stripmining plaats in stukken van 32)




```
CVD$ NODEPCHK
```

Niets nieuws onder de zon dus. Wel nieuw is dat parallelle verwerking van een DO-loop kan worden afgedwongen door CVD$ CONCUR voor die loop te plaatsen. Binnen zo'n loop kunnen dan eventueel weer loops gevectoriseerd worden.

Al met al geeft dit ons veel keuzemogelijkheden waardoor een keuze niet eenvoudig is. Bij onze versie van het verdeel-en-heers algoritme (paragraaf 8.3 en paragraaf 13.3) hebben we in de eerste fase de keuze tussen het parallel uitvoeren van de recursies

voor de subgroepen (paragraaf 13.3) en het in opeenvolgende slagen simultaan uitvoeren van de recursie voor de overeenkomstige elementen van elke subgroep door middel van een vectoroperatie (paragraaf 8.3.c). In het laatste geval moeten de vectoroperaties echter in de goede volgorde plaats vinden en eventueel extra parallellisme kan alleen gevonden worden door de vectoroperaties zelf op te splitsen. Dit is natuurlijk alleen lonend als die vectoroperaties lang genoeg zijn; voor korte vector loops (een klein aantal subgroepen dus) zal de parallelle variant uit paragraaf 13.3 wellicht de voorkeur verdienen.

We geven als voorbeeld twee versies van het verdeel-en-heersalgoritme, één volgens de parallelle aanpak en één volgens de vectoraanpak.

A. Het verdeel-en-heersalgoritme volgens de aanpak in paragraaf 13.3:

```
      SUBROUTINE WANGP1(K,M,A,B)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION A(K,M), B(K,M)
CVD$  CONCUR
      DO 20 J = 1,M
        DO 10 I = 2,K
          B(I,J) = B(I,J) + A(I,J) * B(I-1,J)
          A(I,J) = A(I,J) * A(I-1,J)
 10     CONTINUE
 20   CONTINUE
C   DE TWEEDE FASE: HET CORRIGEREN DER ELEMENTEN
      DO 40 J = 2,M
CVD$  NODEPCHK
        DO 30 I = 1,K
          B(I,J) = B(I,J) + A(I,J) * B(K, J-1)
 30     CONTINUE
 40   CONTINUE
      RETURN
      END
```

Toelichting:

1. Het berekenen van de correctiecoëfficiënten in de DO 10-loop is overbodig voor j = 1; op deze manier echter sparen we een if-statement uit en wordt het vectoriseren eenvoudiger.
2. De DO 30-loop kan beter vervangen worden door aanroep van een BLAS-routine:

```
CALL DAXPY(K,B(K,J-1),A(1,J),1,B(1,J),1)
```

B. Het verdeel-en-heersalgoritme volgens de beschrijving in paragraaf 13.3.c, met aanpak II:

```
      SUBROUTINE WANGV2(K,M,A,B)
      IMPLICIT DOUBLE PRECISION (A-H, 0-Z)
      DIMENSION A(K,M), B(K,M)
      DO 20 I = 2,K
CVD$  NODEPCHK
        DO 10 J = 1,M
          B(I,J) = B(I,J) + A(I,J) * B(I-1,J)
          A(I,J) = A(I,J) * A(I-1,J)
 10     CONTINUE
 20   CONTINUE
C   CORRECTIE LAATSTE ELEMENT VAN ELKE SUBGROEP:
      DO 30 J = 2,M
        B(K,J) = B(K,J) + A(K,J) * B(K, J-1)
 30   CONTINUE
CVD$  CONCUR
      DO 40 J = 2,M
        CALL DAXPY(K-1,B(K,J-1),A(1,J),1,B(1,J),1)
 40   CONTINUE
      RETURN
      END
```

Analyse van de snelheid van vector loops kan plaatsvinden met de technieken die in paragraaf 6.2 beschreven zijn. Parallel uitgevoerde loops kunnen geanalyseerd worden als in paragraaf 12.3 en met name bij uitgesplitste vector loops moet men rekening houden met de doorwerking van de 'halve vectorsnelheids'-waarde $n_{\frac{1}{2}}$ in de 'halve parallelle snelheids'-waarde $n_{\frac{1}{2},p}$: $n_{\frac{1}{2},p} > pn_{\frac{1}{2}}$. Bij wijze van voorbeeld zullen we trachten hiermee resultaten uit [42] te herinterpreteren. Zoals blijkt uit deze resultaten zijn de volgende snelheden geobserveerd voor het inprodukt.

n	p = 1	p = 2	p = 4	p = 8
1000	3.8 Mflops	7.2 Mflops	13.6 Mflops	23.0 Mflops
5000	3.9 Mflops	7.5 Mflops	14.8 Mflops	27.7 Mflops

We moeten er wel rekening mee houden dat slechts twee metingen voor elke p een wankele basis zijn voor het bepalen van de snelheidsparameters. Voorts hebben we er geen idee van hoe nauwkeurig de metingen zijn.

Met behulp van de formule

$$R(\tilde{A},n,p) = R_\infty(\tilde{A},p)\ \frac{n}{n+n_{\frac{1}{2},p}}, \text{ met } n_{\frac{1}{2},1} = n_{\frac{1}{2}},$$

krijgen we de volgende parameterwaarden:

$$R_\infty(\tilde{A},1) = R_\infty(A) \approx 3.9 \qquad n_{\frac{1}{2}} \approx 33$$
$$R_\infty(\tilde{A},2) \approx 7.6 \qquad n_{\frac{1}{2},2} \approx 53$$
$$R_\infty(\tilde{A},4) \approx 15.1 \qquad n_{\frac{1}{2},4} \approx 113$$
$$R_\infty(\tilde{A},8) \approx 29.2 \qquad n_{\frac{1}{2},8} \approx 269$$

We merken op dat $R_\infty(\tilde{A},p) \approx p\,R_\infty(\tilde{A})$ en dat de $n_{\frac{1}{2},p}$-waarden ook ruwweg evenredig zijn met p. Deze uitkomsten doen vermoeden dat de parallelle effectiviteit van de Alliant beter is dan op grond van de meetwaarden in [42] geconcludeerd wordt. Er zijn uiteraard nauwkeuriger meting nodig om dat te bevestigen.

Met de in paragraaf 6.3 beschreven aanpak hebben we op de met ACE's uitgeruste ALLIANT FX/80 de volgende waarden voor het inprodukt gemeten:

$R_\infty(A,1) = 5.07$ Mflops $\quad n_{\frac{1}{2},1} = 18$
$R_\infty(A,2) = 10.18$ Mflops $\quad n_{\frac{1}{2},2} = 78$
$R_\infty(A,4) = 20.38$ Mflops $\quad n_{\frac{1}{2},4} = 160$
$R_\infty(A,8) = 40.07$ Mflops $\quad n_{\frac{1}{2},8} = 341$

We zien hier weer ruwweg de eerder opgemerkte evenredigheid met p, behalve dan dat $n_{\frac{1}{2},1}$ lager is dan op grond van evenredigheid met p verwacht zou worden.

Zoals we in paragraaf 7.2 zagen, is voor de verwerking van het inprodukt slechts beperkte toegang tot het geheugen vereist. Misschien is dat wel een sleutel tot de verklaring voor het feit dat de hierboven opgemerkte evenredigheid voor sommige andere vectoroperaties minder fraai uitpakt. We geven de meetwaarden voor R_∞, behaald op de FX/80, voor de ijle matrix-vectorvermenigvuldiging (loop nummer 14 in paragraaf 6.3, we duiden het algoritme hierna aan met B):

$R_\infty(B,1) = 3.1$ Mflops
$R_\infty(B,2) = 6.2$ Mflops
$R_\infty(B,4) = 12.2$ Mflops
$R_\infty(B,6) = 15.7$ Mflops
$R_\infty(B,8) = 18.0$ Mflops

Uit de gemeten snelheden voor het inprodukt zien we dat voor ideale situaties inderdaad een versnelling met een factor p gehaald wordt ten gevolge van verwerking op p processoren. De vraag is hoe zoiets uitpakt voor meer realistische problemen. We kunnen daar uiteraard geen definitief antwoord op geven, maar kunnen wel een voorlopige indruk verschaffen met de volgende resultaten.

Voor het oplossen van een groot ijl stelsel vergelijkingen hebben we de iteratieve gepreconditioneerde geconjugeerde gradiënten methode (ICCG(0)) gebruikt. Deze methode is zoveel mogelijk in vectoriseerbare operaties uitgeschreven (zie voor details [57, sectie 4.3]). Voor een relatief klein stelsel ($n \approx 3600$) leverde dat de volgende snelheidswaarden op:

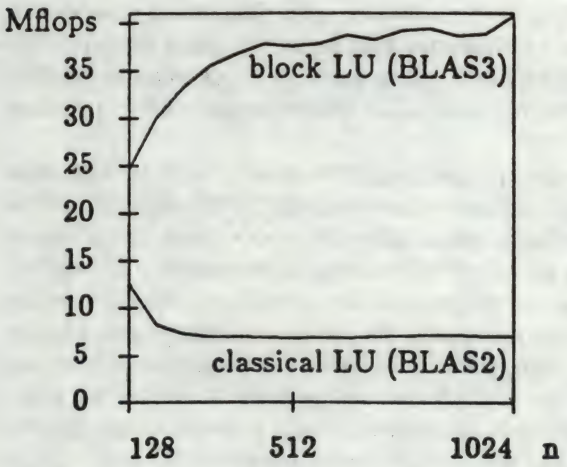
p = 1: 1.97 Mflops
p = 2: 2.57 Mflops
p = 4: 3.45 Mflops.

Voor een zeer veel groter stelsel (n = 40000) werd gemeten:

p = 1: 1.21 Mflops
p = 2: 2.11 Mflops
p = 4: 3.45 Mflops.

Tot onze verbazing zien we dat voor p = 1 de snelheid zakt niettegenstaande de enorm toegenomen vectorlengte. De verklaring moet waarschijnlijk gezocht worden in het feit dat de beperkte grootte van het cache geheugen efficiënte verwerking van lange loops wat tegenwerkt. Dat voor p = 4 de snelheid inmiddels weer op hetzelfde niveau is beland als voor het kleinere probleem komt misschien doordat de langere vectorlengten wel een betere parallelle verwerking toelaten hetgeen minder afhankelijk is van het cache geheugen.

Het effect van het cache geheugen (denk ook aan onze verhandelingen over hiërarchische geheugens in verband met virtual memory en paging in paragraaf 10; bij een cache geheugen vindt iets soortgelijks plaats) treedt ook naar voren bij de matrixvectorvermenigvuldiging volgens de standaard (BLAS of BLAS-2) aanpak. Van Kats en Van der Steen [42] rapporteren voor de kolomsgewijze aanpak de volgende snelheden (p = 4): n = 200: 7.8 Mflops en n = 300: 7.1 Mlops. De degradatie in snelheid wordt door hen ook geweten aan de beperkte grootte van het cache geheugen. Dat met de bloksgewijze BLAS-3 aanpak een veel effectiever geheugenbeheer verkregen kan worden, wordt heel treffend door Jalby [35] geïllustreerd aan de hand van de LU-ontbinding van een volle matrix. Voor p = 8 zijn de snelheden weergegeven in onderstaande grafiek.



Voor de klassieke kolomsgewijze aanpak (BLAS-2) zien we voor toenemende n weer een afnemende snelheid conform de metingen van Van Kats en Van der Steen, echter de bloksgewijze BLAS-3 aanpak (zie ook paragraaf 9) heeft daar in het geheel geen last van en laat een orde betere prestaties toe (merk op dat het aantal flops voor beide aanpakken gelijk is).

14.3 Parallelle supercomputers

Echt grensverleggende rekensnelheden komen in zicht wanneer een aantal CPU's, die elk de kracht van een supercomputer hebben, parallel gebruikt kunnen worden. Er zijn inmiddels al een paar van dit soort rekengiganten, we noemen de IBM 3090 met meerder CPU's + VF, de ETA-10 serie (een parallelle opvolger van de CYBER 205), de CRAY-2 (4 krachtige processoren). De bekendste en oudste machine van dit type is echter wel de CRAY X-MP. Deze laatste computer kent drie basismodellen: een enkele processor-machine CRAY X-MP/1, een dubbele processor-machine CRAY X-MP/2 en de vier processoren-variant CRAY X-MP/4. Elk van deze processoren kan men zien als een verbeterde CRAY-1 computer en is ook als zodanig zelfstandig te gebruiken (zie paragraaf 5).

We zullen het hier uitvoerig hebben over het parallelle gebruik van een aantal van deze krachtige CPU's. Anders dan bij de in de paragrafen 14.1 en 14.2 behandelde parallelle rekenvormen is het hier niet mogelijk, of zelfs wenselijk, om gedurende de uitvoering van een geheel programma over een vast aantal processoren te beschikken (tenzij men natuurlijk de gehele machine voor zichzelf reserveert). Men laat het besturingssysteem op ieder moment beslissen hoeveel processoren men krijgt toegewezen op de punten waar men in een programma heeft aangegeven er meerdere te kunnen gebruiken. Dit laatste gebeurt als volgt.

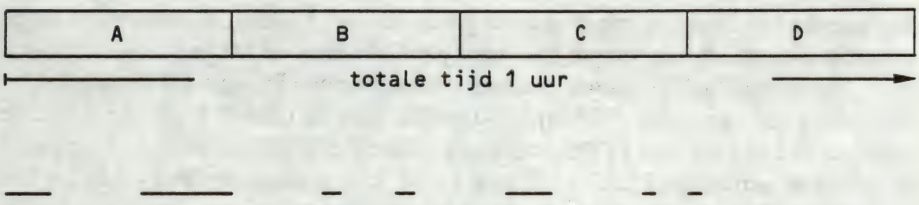
Op de momenten dat er voldoende parallel werk is wordt dit parallelle werk in de vorm van zogenaamde tasks aan het systeem aangeboden, in de inputqueue voor alle CPU's geplaatst en elke CPU kijkt volgens zekere voorrangsregels steeds in de inputqueue of er wat te rekenen valt. Alle tasks doen dus mee in het programma-aanbod voor het gehele systeem en men zorgt er op die manier voor dat men in plaats van bij 1 CPU nu bij alle CPU's in de rij staat. Men kan daardoor bereiken dat de totale verblijftijd binnen het systeem aanzienlijk afneemt zonder dat de totale rekentijd voor de gebruikte CPU's veel groter wordt.

Het is nu wel zaak dat men geen parallelle algoritmen kiest die beduidend meer flops kosten dan de beste seriële algoritmen omdat er uiteindelijk (doorgaans) voor de door alle CPU's gezamelijk verspijkerde rekentijd betaald moet worden. Om er voor te zorgen dat de verschillende tasks, die op vrij willekeurige momenten door de beschikbare CPU's verwerkt worden, goed op elkaar aansluiten, worden vrij hoge synchronisatiekosten gemaakt (zie ook [32]) en men doet er dus goed aan alleen vrij grote klussen als task aan te bieden om deze kosten relatief te drukken. Het parallellisme moet dus vooral gezocht worden op het niveau van tijdvretende subroutine aanroepen, het zogenaamde macrotasking. Bij het genereren van tasks hoeft men geen rekening te houden met het fysiek aanwezige aantal CPU's. Door meer tasks te genereren draagt men bij tot een betere verdeling van de totale werklast over de CPU's en is men in principe klaar voor gebruik van meer CPU's zodra die aan het systeem worden toegevoegd.

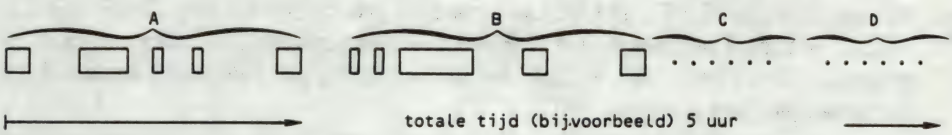
Sinds kort is er ook een veel kleinschaliger vorm van parallellisme beschikbaar op DO-loop niveau, het zogenaamde microtasking. We zullen daar nog op terugkomen.

We lichten het effect van macrotasking toe aan de hand van een fictief voorbeeld voor een eveneens fictieve vier processor-machine. We nemen aan dat de verwerking

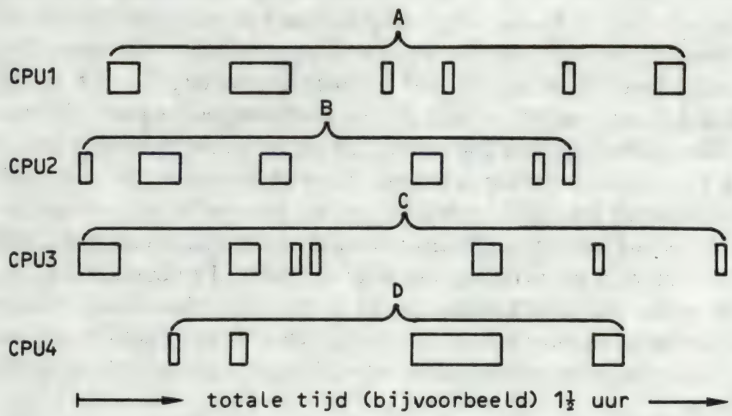
van het programma bestaat uit vier even grote onderdelen A, B, C en D, die desgewenst parallel uitvoerbaar zijn. Indien de uitvoering van het gehele programma op 1 CPU ongeveer een uur aan zuivere rekentijd kost, dan zal de verblijftijd doorgaans veel groter zijn vanwege het timesharing mechanisme, dat ervoor zorgt dat een aantal programma's om beurten een portie tijd krijgt toegemeten, zodat bijvoorbeeld tijdens het plegen van relatief trage I/O er geen kostbare rekentijd verloren gaat (I/O wordt verzorgd door aparte (goedkopere) processoren). Voor uitvoering op één lege CPU zouden we schematisch in de tijd gezien het volgende beeld krijgen (één blok geeft de tijd voor een onderdeel aan, gedurende welke het systeem onafgebroken met dat onderdeel bezig is):



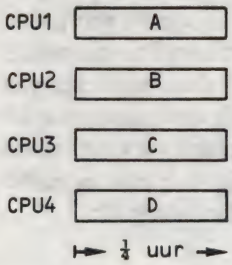
Bij verwerking op 1 CPU die een beladen programma-aanbod heeft, ziet de verwerking er schematisch uit als:



Bij multitasking van de onderdelen A, B, C en D op een druk bezet systeem krijgen we dan iets in de geest van:



Op een leeg (gereserveerd) systeem krijgen we met multitasking:



We zien dat het, zowel op een druk bezet als op een leeg systeem, de moeite kan lonen om grote onderdelen van een programma te parallelliseren via het genereren van tasks.

In zijn inmiddels befaamde prestatie-overzicht meldt Dongarra [17] voor de LU-ontbinding van een volle 1000 bij 1000 matrix voor de CRAY X-MP de volgende getallen: 192 Mflops voor 1 CPU (maximale snelheid 210 Mflops) en 713 Mflops voor de parallelle versie (via BLAS-3 [18]) op 4 CPU's (maximale snelheid 4 * 210 = 840 Mflops). Beide snelheden zijn gemeten op een overigens lege machine. De gehaalde versnelling bedraagt dus 3.7. Uit tabel 2 in paragraaf 12.2 zien we dat het inzetten van nog meer van dit soort krachtige CPU's snel minder rendabel wordt (voeg daarbij dan nog het effect van $p\,n_{\frac{1}{2}}$ vanwege het vectoriseren van steeds kortere loops; zie paragraaf 12.3). Voor de IBM 3090 worden de volgende getallen gemeld: 1 processor haalt 71 Mflops (3090/180E VF); 141 Mflops voor de 2 processor IBM 3090/200E VF; 270 Mflops voor de 4 processor IBM 3090/400E VF en tenslotte 387 Mflops voor de 6-processor IBM 3090/600E VF. Ook hier zien we dat bij een toenemend aantal CPU's het rendement snel afneemt.

Voor grote produktiecodes worden, althans voor de CRAY X-MP, soortgelijke versnellingen gemeld. Het European Centre for Medium-range Weather Forecasts (ECMWF) maakt voor haar dagelijkse weersvoorspellingen gebruik van een CRAY X-MP/4. Dent en Gibson [10] beschrijven een aantal interessante details van de produktie-voorspellingscode die sinds 1983, met tussentijdse aanpassingen, in dagelijks gebruik is (aanvankelijk op een CRAY-1). Het zorgvuldig multitasken van deze code leidde ertoe dat circa 90% van het werk effectief parallel verwerkt kon worden (een versnelling van ruim 3). De niet effectief benutte 10% was te wijten aan het ongelijk zijn van de geparallelliseerde tasks (ongeveer 5% verlies), synchronisatiekosten (ongeveer 4% verlies) en een stuk seriële code (ongeveer 2% verlies). De oerversie van deze code nam in 1979 op een CRAY-1 ongeveer vier uur in beslag. De nieuwste versie, die uitgaat van een mathematisch verfijnder (en daardoor duurder) model, met meer niveaus in verticale richting en een beduidend betere resolutie in horizontale richtingen, neemt sinds 1986 op de CRAY X-MP/4 circa $2\frac{3}{4}$ uur in beslag (dus beter en sneller). De door Dent en Gibson getoonde overeenkomsten tussen de voorspelde en de opgetreden actuele situatie zijn, althans voor een leek, indrukwekkend. Recentelijk heeft men nog kans gezien het parallellisme in de code verder op te voeren door het werk beter in gelijke tasks te verdelen (load-balancing).

Hockney [32] geeft een overzicht van de parallelle programmeertechnieken voor

de CRAY X-MP en meldt ook voor zeer eenvoudige constructies gemeten waarden voor de synchronisatie kosten. We kregen daarbij niet de indruk dat de doorwerking van de $n_{\frac{1}{2}}$-waarde (paragraaf 12.3) opgemerkt werd.

Wij zullen, louter ter illustratie, ook een tweetal voorbeelden geven van macrotasking voor een zeer eenvoudig voorbeeld. Evenmin als Hockney zouden we willen suggereren dat we in de praktijk voor dit soort gevallen multitasking, in de vorm van macrotasking, van stal zouden moeten halen.

In het voorbeeld gaan we uit van de vector update of SAXPY (paragraaf 7.1):

```
      DO 10 I = 1,N
        X(I) = X(I) + ALFA * A(I)
10    CONTINUE
```

en we verdelen dit werk in vier min of meer gelijke porties. Eén van die porties laten we in het hoofdprogramma verwerken; de overige drie worden via tasks aan de beschikbare CPU's aangeboden. We laten eerst het eenvoudigste mechanisme zien (Hockney duidt dat aan met de naam TASKS).

a. TASKS

Het relevante deel van het programma, dat de bovenstaande DO-loop vervangt, komt er ongeveer als volgt uit te zien:

```
CALL TSKSTART(IDT1,TRIAD,N/4 + 1,N/2,X,A,ALFA)
```

(toelichting: IDT1 is een integer array met de lengte 2 waarin het systeem identificatie van de task bijhoudt; TRIAD is de naam van de subroutine die de vector update in dit geval verzorgt en de overige parameters zijn de parameters (actuele) van TRIAD; deze aanroep vervangt dus de aanroep `CALL TRIAD(N/4 + 1,N/2,X,A,ALFA)`; Deze aanroep van TSKSTART genereert een TASK voor de uitvoering van het tweede segment van de vector update, het derde en vierde segment worden als volgt behandeld:)

```
CALL TSKSTART(IDT2,TRIAD,N/2 + 1,3 * N/4,X,A,ALFA)

CALL TSKSTART(IDT3,TRIAD,3 * N/4 + 1,N,X,A,ALFA)
```

(toelichting: nadat deze drie tasks gegenereerd zijn laten we het hoofdprogramma zelf verder gaan met de rest van de update:)

```
      DO 10 I = 1,N/4
        X(I) = X(I) + ALFA * A(I)
10    CONTINUE
```

(toelichting: nadat we alle tasks gegenereerd hebben, vervolgt het hoofdprogramma zijn eigen weg, echter nadat het de DO-loop voltooid heeft is het nog niet zeker of de andere

tasks inmiddels ook al afgewerkt zijn. Wil men daar voor het verdere verloop van het programma zeker van zijn, dan kan men de CRAY routine TSKWAIT gebruiken die informatie uit de task arrays IDT haalt:)

```
CALL TSKWAIT(IDT1)
```

(het programma gaat pas verder als TASK 1 klaar is)

```
CALL TSKWAIT(IDT2)
```

(dient voor de tweede TASK)

```
CALL TSKWAIT(IDT3)
```

(nadat ook de derde TASK gereed is, wordt het hoofdprogramma verder voortgezet)

In dit geval ziet de door de gebruiker mee te leveren subroutine TRIAD er bijvoorbeeld als volgt uit:

```
      SUBROUTINE TRIAD(N1,N2,X,A,ALFA)
      DIMENSION X(*), A(*)
      DO 10 I = N1, N2
        X(I) = X(I) + ALFA * A(I)
10    CONTINUE
      RETURN
      END
```

b. EVENTS

Het kan voorkomen dat parallel uit te voeren processen tussentijds op elkaar moeten wachten totdat de juiste data weggeschreven zijn of om elkaar de kans te geven nog data te gebruiken voordat ze overschreven worden. Dat vraagt om een fijnzinnig synchronisatiemechanisme en het TASKS-mechanisme dat we zojuist beschreven is daarvoor te grof. Het zogenaamde EVENTS-mechanisme is meer toegesneden op onderlinge synchronisatie van acties. Het EVENTS principe is gebaseerd op het zetten van 'vlaggetjes' (de zogenaamde events), die de status van een berekening kunnen aangeven. We zullen het principe schetsmatig toelichten aan de hand van het al eerder besproken verdeel-en-heersalgoritme. Vooreerst merken we weer op dat dit algoritme in zijn parallelle vorm weinig geschikt is voor verwerking op parallelle vector CPU's, vanwege de recursie in de binnenste loop van de verdeelfase en vanwege de algehele kleinschaligheid van het algoritme. Voor een bespreking van het verdeel-en-heersalgoritme: zie de paragrafen 8.3.c, 13.3 en 14.2.

De parallelle aanpak van het algoritme zou als volgt kunnen geschieden. We laten elke task bestaan uit de berekeningen die horen bij één subgroep van de totale recursie. In de eerste verdeelfase is er nog niets aan de hand en kunnen de recursies voor de sub-

groepen parallel worden uitgevoerd. Daarna treedt de correctiefase in (fase 2). Echter voor correctie van de j-de subgroep moet de betreffende task eerst beschikken over de definitieve (gecorrigeerde) waarde van het laatste element van de (j-1)-ste subgroep. We zetten fase 2 daarom in door eerst die laatste elementen te berekenen (het betreft hier in feite de recursie in het tweede programma in paragraaf 14.2). De recursie hoeft evenwel niet voltooid te zijn, alvorens de diverse tasks aan het werk kunnen gaan. Zodra een laatste element bekend is, kan de task voor de volgende subgroep meteen aan de slag.

We hebben ons in het voorbeeld maar weer beperkt tot vier subgroepen. Er zijn speciale synchronisatievariabelen nodig, we zullen ze voor het gemak maar vlaggetjes noemen en deze vlaggen moeten in COMMON geplaatst worden. Neem maar aan dat ze in het hoofdprogramma in het COMMON met label /VLAG/ staan. Deze variabelen zullen we aangeven met INij, waarbij i en j cijfers zijn: i slaat op de i-de fase (i = 1 of 2), j slaat op de subgroep (j = 1, 2, 3 of 4). Fase 1 zullen we rekenen op te houden bij de correctie van het laatste element van een subgroep. Na al deze inleidende afspraken, kan het hele algoritme met de EVENTS-aanpak er schematisch als volgt uit komen te zien.

```
      COMMON/VLAG/IN11, IN12, IN13, IN22, IN23, IN24
      . . . . .
      . . . . .
      . . . . .
C     WE MOETEN NU AANGEVEN DAT DE VARIABELEN INij BEDOELD ZIJN
C     ALS SYNCHRONISATIE VARIABELEN EN ZE ALLEMAAL 'NAAR
C     BENEDEN' LATEN WIJZEN

      CALL EVASGN(IN11)
      CALL EVASGN(IN12)
      CALL EVASGN(IN13)
      CALL EVASGN(IN22)
      CALL EVASGN(IN23)
      CALL EVASGN(IN24)
C     VERVOLGENS, AANGENOMEN DAT ALLE VARIABELEN EN ARRAYS HUN
C     WAARDEN HEBBEN GEKREGEN, WORDEN DE TASKS OPGESTART:
      CALL TKSTART (IDT1,SOLVE,N,2,A,B)
      CALL TKSTART (IDT2,SOLVE,N,3,A,B)
      CALL TKSTART (IDT3,SOLVE,N,4,A,B)
C     HET HOOFDPROGRAMMA VERVOLGT DE WERKZAAMHEDEN AAN DE
C     EERSTE SUBGROEP:
      DO 10 I = 1, N/4
        B(I) = B(I) + A(I) * B(I-1)
10    CONTINUE
C     NA AFLOOP HIERVAN IS DE EERSTE FASE VOOR SUBGROEP 1
C     AFGESLOTEN EN KAN 'VLAG' IN11 WORDEN UITGESTOKEN:
      CALL EVPOST (IN11)
C     NU MOETEN WE WACHTEN TOTDAT DE TASKS HUN WERK AAN DE
C     OVERIGE SUBGROEPEN VOLTOOID HEBBEN. DAT WORDT AANGEGE-
```

```
C   VEN DOORDAT DE VLAGGEN BEHOREND BIJ DE 2-DE FASE VAN DE
C   2-DE, DE 3-DE EN DE 4-DE SUBGROEPEN ZIJN UITGESTOKEN.
C   DAT WACHTEN GEBEURT VIA:
        CALL EVWAIT(IN22)
        CALL EVWAIT(IN23)
        CALL EVWAIT(IN24)
C   HET VERDEEL-EN-HEERSALGORITME IS NU VOLTOOID EN HET
C   PROGRAMMA KAN MET ANDER WERK VERDER GAAN.
        . . . . .
        . . . . .
```

Zoals we gezien hebben, gebeurt het eigenlijke werk van elke task in feite in de subroutine SOLVE die al het werk aan een subgroep uitvoert. Deze subroutine SOLVE kan er als volgt uitzien:

```
        SUBROUTINE SOLVE (N,K,A,B)
        DIMENSION A(*), B(*)
        COMMON/VLAG/ IN11, IN12, IN13, IN22, IN23, IN24

C   SOLVE VOERT HET REKENWERK UIT VAN HET VERDEEL-EN-HEERS-
C   ALGORITME VOOR DE K-DE SUBGROEP. WE BEPALEN EERST DE
C   BEGIN- EN EIND-INDEX VAN DEZE SUBGROEP:
        N1 = ((K -1) * N) / 4 + 1
        N2 = (K * N) / 4
C   FASE 1 VOOR DE K-DE SUBGROEP
        DO 10 I = N1 + 1, N2
          B(I) = B(I) + A(I) * B(I-1)
          A(I) = A(I) * A(I-1)
 10     CONTINUE
C   NU MOET DE VERDERE BEREKENING WACHTEN TOTDAT HET LAATSTE
C   ELEMENT VAN DE (K-1)-STE SUBGROEP ZIJN DEFINITIEVE WAAR-
C   DE HEEFT
        IF (K.EQ.2) CALL EVWAIT(IN11)
        IF (K.EQ.3) CALL EVWAIT(IN12)
        IF (K.EQ.4) CALL EVWAIT(IN13)
C   MET BEHULP VAN HAT LAATSTE ELEMENT VAN DE (K-1)-STE SUB-
C   GROEP, B(N1-1), CORRIGEREN WE EERST HET LAATSTE ELEMENT
C   VAN DE K-DE SUBGROEP EN STEKEN METEEN DE BETREFFENDE VLAG
C   UIT:
        BB = B(N1-1)
        B(N2) = B(N2) + A(N2) * BB
        IF (K.EQ.2) CALL EVPOST(IN12)
        IF (K.EQ.3) CALL EVPOST(IN13)
C   DAARNA KAN HET OVERIGE CORRECTIEWERK VAN FASE 2 PLAATS-
C   VINDEN;
```

```
      DO 20 I = N1, N2-1
        B(I) = B(I) + A(I) * BB
 20   CONTINUE
C   BOVENSTAANDE OPDRACHT KAN OOK VERVANGEN WORDEN DOOR EEN
C   AANROEP VAN SAXPY (HOOFDSTUK 7).
C   TENSLOTTE WORDT DE VLAG UITGESTOKEN TEN TEKEN DAT DE
C   TWEEDE FASE KLAAR IS:
      IF (K.EQ.2) CALL EVPOST(IN22)
      IF (K.EQ.3) CALL EVPOST(IN23)
      IF (K.EQ.4) CALL EVPOST(IN24)
      RETURN
      END
```

De volgende kanttekeningen kunnen nog gemaakt worden:

- afgezien van het feit dat het algoritme niet zeer geschikt is voor parallelle verwerking op meervoudige vector CPU's, is de werklast bovendien ook niet erg evenwichtig verdeeld. De tasks hebben ongeveer twee keer zoveel te doen als het hoofdprogramma;
- als men wacht totdat een bepaalde vlag uitgestoken is, maar men vergeet dat uitsteken te doen in de betreffende task, dan treedt de beruchte deadlock situatie op. Het systeem blijft wachten, zich steeds afvragend of het betreffende vlaggetje inmiddels al uitgestoken is. Dit afvragen kost gelukkig (?) ook wat tijd zodat het programma na het opsouperen van de opgegeven CPU-tijd (CP-TIME LIMIT) beëindigd wordt en we de gelegenheid krijgen te zien wat er gebeurd is.

De CRAY X-MP kent ook een veel fijnschaliger manier om parallellisme aan te geven: het zogenaamde microtasking. Dit gebeurt weer via zogenaamde Comment Directives en vertoont grote gelijkenis met de gang van zaken bij de Alliant FX/8. De verschillen bestaan hieruit dat elke parallel uit te voeren loop expliciet aangegeven moet worden (zoals bij SEQUENT) en dat we niet zeker weten of er inderdaad ook echt iets parallel wordt uitgevoerd. Er worden in feite alleen kleinschalige tasks gegenereerd en dus alleen als men de beschikking heeft over een lege (dedicated) machine is men zeker van echte parallelle verwerking.

Voor een te microtasken DO-loop plaatst men de directive

```
CMIC$ DOGLOBAL
```

Subroutines waarin microtasking wordt aangewend via CMIC$-directieven, moeten op een speciale wijze gecompileerd worden, met het oog op automatische synchronisatie, en dat wordt aangegeven door onmiddellijk voorafgaand aan de SUBROUTINE declaratie de directive

```
CMIC$ MICRO
```

te plaatsen.

CRAY beweert dat de synchronisatie-overhead bij microtasking tamelijk gering

is, namelijk nooit meer dan 5%. Alle details omtrent microtasking kan men vinden in de betreffende CRAY handleiding [7].

Het multitasking principe is hier wat uitgebreider aan de orde gekomen omdat het, zij het in varianten, in principe net zo wordt toegepast bij de andere multiprocessor supercomputers, zoals de CRAY-2, de IBM 3090-serie, de ETA-10 serie [25], de CRAY Y-MP [8] en de CONVEX C200 serie.

VIJFTIEN

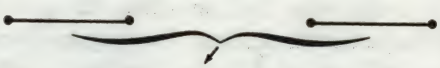
MESSAGE PASSING SYSTEMEN

In een message passing systeem heeft elke processor een eigen geheugen en vindt commmunicatie plaats door het expliciet verzenden van boodschappen via de onderlinge verbindingen. Omdat het bij een groter aantal processoren niet mogelijk is ze allen met elkaar te verbinden, moet men een keuze maken. Het (commercieel) meest succesvolle systeem is de zogenaamde Hypercube structuur. Een *hypercube* (we houden het Engelse woord maar aan) van orde n ontstaat door de overeenkomstige punten van twee hypercubes van orde n-1 met elkaar te verbinden. Een hypercube van orde 0 is één enkel punt (of node, of, in ons geval, een processor).

Met behulp van deze definitie kunnen we de hypercubes van lage orde als volgt representeren (een . stelt een processor voor en een lijnstuk de verbinding tussen twee processoren):

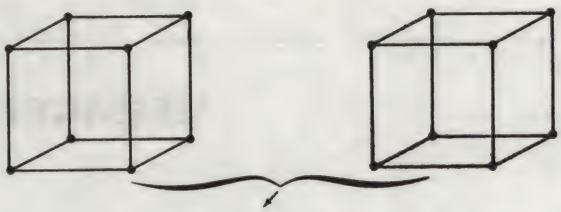
Twee hypercubes van orde 0:

Twee hypercubes van orde 1:

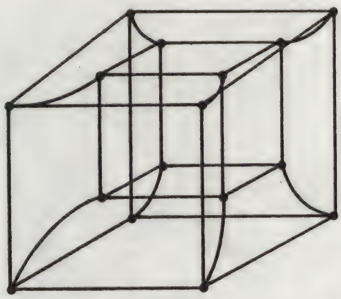


Twee hypercubes van orde 2:
(4 nodes)

Twee hypercubes van orde 3:
(8 nodes)



Hypercube van orde 4:
(16 nodes)



Merk op dat een hypercube systeem van orde n uit 2^n processoren bestaat, dat elke processor met precies n andere processoren verbonden is en dat de maximale 'afstand' tussen twee processoren uit n verbindingslijnen bestaat. Met een betrekkelijk gering aantal verbindingen per processor kan men dus een systeem met een flink aantal processoren in elkaar zetten. Een paar representanten van de huidige systemen zijn:

Ametek System 14	: maximale orde 8 : $\leq$ 256 nodes
FPS-T Series	: maximale orde 14 : $\leq$ 16 384 nodes
	(het grootste geleverde systeem volgens [16] : 32 nodes)
Intel's iPSC	scalaire nodes : maximale orde 7 : $\leq$ 128 nodes
	vector nodes : maximale orde 6 : $\leq$ 64 nodes
NCUBE	: maximale orde 10 : $\leq$ 1024 nodes

We zullen in onze behandeling van de programmering van message passing systemen geen rekening houden met de onderlinge verbindingsstructuur. Men zal zich kunnen voorstellen dat het voordeel kan hebben de communicatie tussen processoren zoveel mogelijk te beperken tot naburen. Voor de gebruiker is dat meestal afgeschermd en zorgt het systeem zelf voor transport via de optimale verbindingsweg. Het ziet er voor de programmeur dus uit alsof alle processoren onderling met elkaar verbonden zijn. In gevallen waarin transport van data tussen de processoren een dominerende rol speelt kan men evenwel beter proberen rekening te houden met een zekere verbindingsstructuur en trachten een algoritme zo te organiseren dat lange communicatieroutes vermeden worden. Er zijn ons op dit moment nog geen publicaties bekend waarin dit systematisch wordt onderzocht.

De communicatie met de buitenwereld en het opstarten van programma's gebeurt via een zogenaamde host, dat is meestal één van de nodes (node 0), het kan echter ook een aparte computer zijn (bijvoorbeeld een pc, zoals bij de NCUBE/4). Hiervan kan het

nog afhangen of de host zelf wordt ingeschakeld bij de uitvoering van het rekenwerk; we zullen verder aannemen dat de host één van de nodes is of een met de nodes vergelijkbare processor (wat betreft snelheid) en dat de host dus ook een gedeelte van het werk kan uitvoeren.

In [38] geeft Karp voorbeelden van de verschillende programmeerstijlen voor de diverse computersystemen. We nemen zijn voorbeeld voor het berekenen van de som van n arrayelementen A(i) over. Voor een algemeen message passing systeem kan die sommatie er bijvoorbeeld als volgt uitzien (met < > geven we specifieke constructies aan die er per machine verschillend uit kunnen zien):

programma voor de host:

```
      PARAMETER (N = 1000000)
      DIMENSION A(N)
      READ (*) A
      INC = (N + nprocs - 1)/nprocs
```

(nprocs is een machine afhankelijke grootheid die het aantal nodes aangeeft, we nemen aan dat nprocs << N.)

```
      IS = 1
      DO 10 J = 1, nprocs - 1
      < send (J, 'INC:1, A(IS):INC'>
```

(toelichting: bedoeld wordt: zend naar node j de waarde van INC (1 woord) en de waarden A vanaf adres IS (INC woorden).)

```
           IS = IS + INC
10    CONTINUE
      CALL SUMSUB(SUM,A(IS),N - IS + 1)
```

(de host doet de rest)

```
      DO 20 J = 1, nprocs - 1
      < receive from any ('SUMJ:1') >
```

(toelichting: ontvang de boodschappen, bestaande uit slechts één partiële som SUMJ, in de volgorde waarin ze binnenkomen.)

```
      SUM = SUM + SUMJ
20    CONTINUE
      WRITE (*) SUM
      END
      SUBROUTINE SUMSUB(SUM,A,N)
      DIMENSION A(N)
      SUM = 0.
      DO 10 I = 1,N
           SUM = SUM + A(I)
10    CONTINUE
      RETURN
      END
```

het programma voor de overige nodes:

```
      PARAMETER (N=1000000,INCR=(N+nprocs-1)/nprocs)
      DIMENSION A (INCR)
      < receive (0, 'INC:1, A:INC') >
```

(toelichting: dit is het complement van de send in het host programma, de 0 slaat op host.)

```
      CALL SUMSUB(SUM,A,INC)
      < send (0, 'SUM:1')
      END
      SUBROUTINE SUMSUB(SUM,A,N)
      DIMENSION A(N)
      SUM = 0.
      DO 10 I = 1,N
           SUM = SUM + A(I)
10    CONTINUE
      RETURN
      END
```

Voor een goede verwerking van het bovenstaande programma is het wel nodig dat eerst het host programma op de host en het node programma op alle overige nodes geladen wordt (of is). Daar dient men zelf voor te zorgen. Ook de communicatieboodschappen zelf zijn doorgaans ingewikkelder dan het bovenstaande voorbeeld doet voorkomen; er moeten bijvoorbeeld 'message'-identificaties worden meegegeven. Het echte programma voor een iPSC computer van INTEL ziet er dan ook een slag ingewikkelder uit en we geven het als voorbeeld (met dank aan de makers D. Winter en W. Lioen van het CWI):

```
      PROGRAM HOSTPROG
      INTEGER NODEPID, HOSTPID, ALLNODES
      PARAMETER (NODEPID=1, HOSTPID=1, ALLNODES= -1)
      INTEGER I,J,INC,IS,N,NPROCS,CI,TYPE,CNT,FRNODE,FRPID
      PARAMETER (N = 1000000)
      REAL A(N), SUM, SUMJ
C     Declaratie van de gebruikte iPSC Systeem routines.
      INTEGER COPEN, CUBEDIM
      EXTERNAL COPEN, CUBEDIM

      READ (*) A
C     Laad het programma 'nodeprog' op de nodes:
      CALL LOAD('NODEPROG', ALLNODES, NODEPID)
C     Open een vast kanaal voor de communicatie met alle
C     nodes:
      CI = COPEN(HOSTPID)
      NPROCS = 2**CUBEDIM ()
```

```
        INC = (N + NPROCS - 1)/ NPROCS
       IS = 1
C   Alle data worden naar de nodes verzonden, de gast-
C   heer is zelf geen node:
        DO 10 J = 0, NPROCS - 1
        CALL SENDMSG(CI, 10, A(IS), 4*INC, J, NODEPID)
C   Toelichting: 10 is een message identificatie, 4*INC
C   is de lengte vanhet over te zenden stuk van A in
C   bytes (1 woord = 4 bytes).
           IS =  IS + INC
 10     CONTINUE

        CALL SUMSUB(SUM, A(IS), N-IS + 1)

        DO 20 J = 0, NPROCS - 1
        CALL RECVMSG(CI, TYPE, SUMJ, 4, CNT, FRNODE, FRPID)
C   Toelichting: 4 geeft aan dat de message uit 1 woord
C   (= 4 bytes) bestaat.
           SUM = SUM + SUMJ
 20     CONTINUE
        WRITE (*) SUM
        END
        SUBROUTINE SUMSUB(SUM, A, N)
        (als in het vorige voorbeeld)
```

Het programma dat op de nodes tot uitvoering moet worden gebracht wordt:

```
    PROGRAM NODEPROG
    INTEGER HOSTNID, HOSTPID
    INTEGER HOSTCHAN,OWNNODE,OWNPID,RCNT,RNODE,RPID,INC,
   1        N,NPROCS
    PARAMETER (N = 1000000)
    REAL A(N), SUM
C   Declaratie van gebruikte iPSC routines.
    INTEGER COPEN, MYNODE, MYPID, CUBEDIM
    EXTERNAL COPEN, MYNODE, MYPID, CUBEDIM

    OWNNODE = MYNODE ()
    OWNPID = MYPID ()
C   Hiermee heeft de identificatie van de betreffende node
C   plaatsgevonden.

    NPROCS = 2**CUBEDIM ()
C   Open een vast kanaal voor communicatie met de host:
```

```
HOSTCHAN = COPEN(OWNPID)

CALL RECVW(HOSTCHAN, 10, A, 4*INC, RCNT, RNODE, RPID)
CALL SUMSUB(SUM, A, INC)
CALL SEND(HOSTCHAN, 20, SUM, 4, HOSTNID, HOSTPID)

END

SUBROUTINE SUMSUB(SUM, A, N)
(als in het vorige voorbeeld)
```

Voordat we ingaan op eigen ervaringen met het verdeel-en-heersalgoritme op de NCUBE/4 melden we eerst nog enkele uit de literatuur beschikbare resultaten voor iPSC Hypercube computers om een indruk te geven van wat er al zo bereikt is.

Uit [30] nemen we de volgende gegevens over:
De iPSC/d7 bestaat uit 128 processoren met elk een snelheid van ongeveer 0.03 Mflops (maximaal). De totale snelheid zou dus ongeveer 128 * 0.03 ≈ 3.8 Mflops kunnen bedragen. Voor de LU factorisatie van een volle matrix van orde 2000 werd 3 Mflops gemeten; dat is ongeveer 75% van de topsnelheid. Overigens bleef de probleemgrootte vanwege geheugengrootte's beperkt tot n = 2000.

De iPSC-VX/d4 heeft 16 vectorprocessoren. Elk van deze VX vectorprocessoren haalt voor zeer lange vectorupdates (DAXPY) ongeveer 2.66 Mflops. Voor n = 1200 (maximale matrixgrootte, vanwege het totale geheugen op deze 16 processormachine) haalt de iPSC-VX/d4 voor de LU-decompositie (via de LINPACK routine SGEFA) een snelheid van circa 16 Mflops, hetgeen 37% van de snelheid is die gehaald zou worden als alle 16 processoren lange DAXPY's parallel zouden uitrekenen.

Uit [17] halen we nog de volgende snelheidsmetingen voor de LINPACK routine SGEFA voor n = 1000:

iPSC/d5 (32 processoren): 0.79 Mflops (maximaal 32 * 0.03 ≈ 1 Mflops)
iPSC-VX/d4 (zie boven): 11 Mflops.

We bespreken tenslotte nog onze eigen ervaringen met de NCUBE/4. Het programma daarvoor geven we in de schematische vorm die Karp hanteerde (zie ons eerder gegeven voorbeeld). De organisatie van het programma, voor het verdeel-en-heersalgoritme (de paragrafen 8.3.c en 13.3) loopt vrijwel analoog aan het EVENTS voorbeeld voor de CRAY X-MP (paragraaf 14.2). Het enige verschil is dat hier de data expliciet moeten worden overgezonden en ontvangen. Dat bepaalt automatisch de synchronisatie zodat de 'vlaggetjes' zelf achterwege kunnen blijven.

De relevante delen van het programma op de host worden:

```
PROGRAM HOSTPROG
. . . . . .
. . . . . .
< load nodeprogr on nodes >
```

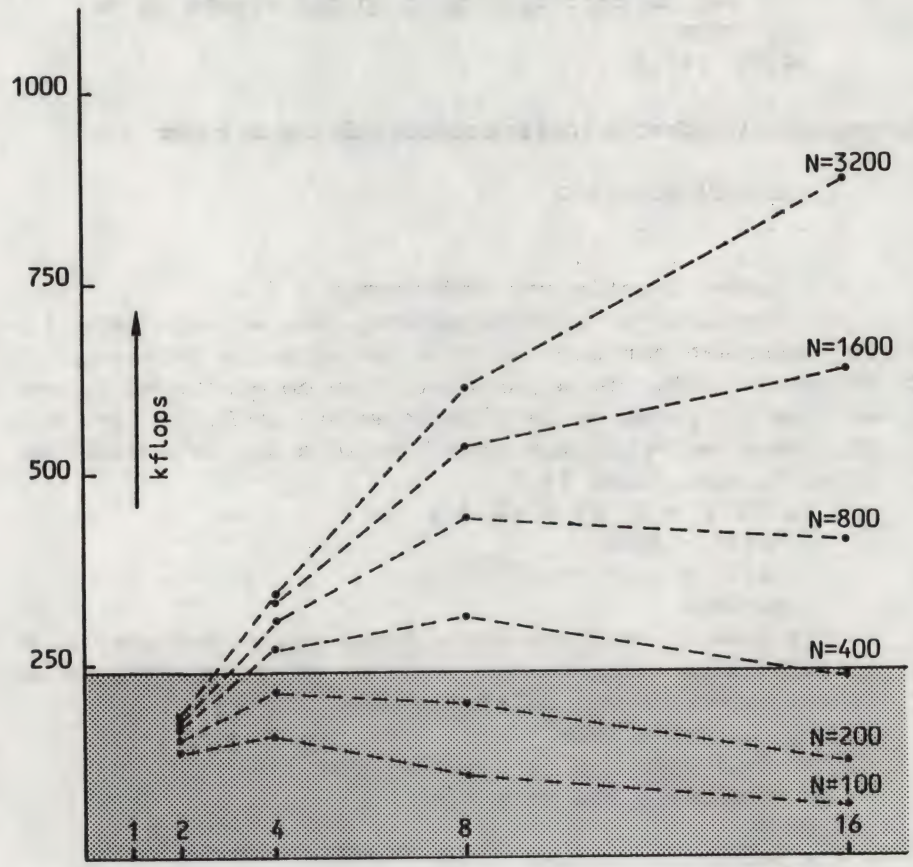
```
      DO 10 J = 0, NPROCS - 1
C   We verdelen het hele werk over alle nodes, elke node
C   krijgt een subgroep; de host doet niet mee aan het
C   werk.
        < zend data van subgroep j naar node j >
 10   CONTINUE
C   Nu worden de door de nodes teruggezonden gedeelten
C   van de oplossing opgevangen en op hun goede plaats
C   in het array A gezet.
      DO 20 J = 0, NPROCS - 1
        < receive data from any node, node id = nodeid >
C   De eerst binnenkomende data hoeven niet noodzakelijk
C   van node 0 te komen, etc., daarom kijken we naar
C   nodeid (het nummer van de node) en plaatsen de data
C   overeenkomstig.
        < zet de ontvangen data op hun plaats in A >
 20   CONTINUE
      WRITE (*) A
```

Het programma voor de nodes komt er schematisch als volgt uit te zien:

```
      PROGRAM NODEPROG
      . . . . . .
      . . . . . .
      < identificatie van nodenummer : j >
      < lees data verzonden door host, bestemd voor node j >
C   We nemen aan dat door de host de volgende gegevens
C   verzonden zijn: de beginindex n1 en de eindindex n2 van
C   de j-de subgroep en de bijbehorende coëfficiënten. Het
C   programma vervolgt dan geheel analoog aan de subroutine
C   SOLVE in paragraaf 14.3:
      DO 10 I = 2, N2 - N1 + 1
        B(I) = B(I)
        A(I) = A(I) * A(I.1)
 10   CONTINUE
C   Node 0 moet nu het laatste element verzenden naar node
C   1. De overige nodes moeten wachten op het gecorrigeerde
C   laatste element van de vorige node:
      IF (J.EQ.0) THEN
      < zend B(N2 - N1 + 1) naar node 1 >
      < zend alle B's naar host >
      STOP
      ELSE
      < lees laatste element van node nummer j-1: BB >
C   Bereken hiermee het laatste element van de j-de subgroep:
```

```
          B(N2 - N1 + 1) = A(N2 - N1 + 1) * BB
        IF (J.NE.NPROCS - 1) THEN
        < zend B(N2 - N1 + 1 naar node j + 1 >
C   Vervolg de correctiefase voor de j-de subgroep.
        DO 20 I = 1, N2 - N1
          B(I) = B(I) + A(I-1) * B(N2 - N1 + 1)
 20     CONTINUE
        < zend B-waarden van subgroep j naar host terug >
        STOP
        END
```

Met behulp van een soortgelijk programma hebben we snelheidsmetingen uitgevoerd op een hypercube processor van het merk NCUBE, namelijk de NCUBE/4. De tijdmetingen zijn gedaan op 'deel'-hypercubes van orde 1, 2, 3 en 4, dus respectievelijk 2, 4, 8 en 16 processoren. De resultaten zijn weergegeven in de grafiek in figuur 14.



Figuur 14

In het gearceerde gebied in figuur 14 was men sneller klaar geweest indien de hele recursie op één enkele node verwerkt was.

Bij de tijdmeting is gemeten vanaf het zenden van de data tot het moment dat de laatste node klaar was. Merk op dat de datacommunicatie tussen de nodes beperkt is gebleven tot het doorgeven van het laatste element van elke subgroep naar de volgende node.

We hebben ook nog geprobeerd het parallelle verdeel- en heers algoritme volgens aanpak I (zie paragraaf 8.3.c) uit te voeren. In dat geval moeten na afloop van fase 1 de data van een subgroep over alle processoren verspreid worden om de correctie voor één subgroep parallel te kunnen uitvoeren. Ten gevolgen van de grote hoeveelheid data die nu verzonden moet worden blijkt het parallelle algoritme volgens deze aanpak op een NCUBE/4 *altijd* trager te zijn dan de hele recursie op één enkele node. In [46] blijkt uit analyse dat de versnellingsfactor van dit volledig parallelle algoritme, volgens aanpak I, altijd begrensd blijft en in feite nauwelijks afhangt van het aantal processoren.

ZESTIEN

DE NABIJE TOEKOMST

In dit slothoofdstuk willen we kort de blik richten op wat ons volgens de fabrikanten in de nabije toekomst te wachten staat. We hebben gezien dat segmentatie van de floating point functional units, het zogenaamde vectorprocessing, gedurende de afgelopen tien jaar tot volle wasdom is gekomen. Vrijwel alle supercomputers en vele minisupers zijn er momenteel mee uitgerust. Een andere veelbelovende ontwikkeling om de snelheid van processoren op te voeren is de introductie van 'wide instruction words', bijvoorbeeld bij de FPS64/60 en MULTIFLOW [16]. Men zou dit kunnen zien als segmentatie van de instructies. Ook andere dan vectoroperaties kunnen profiteren van deze kleinschalige (of fijnkorrelige) vorm van parallellisme.

Introductie van vector functional units in minisupercomputers, zoals de CONVEX C-1, Alliant FX/8 en SCS 40 [16] heeft ertoe geleid dat deze machines in vele gevallen sneller (en goedkoper) zijn dan de inmiddels bijna klassieke mainframes. Sommige fabrikanten spelen hierop in door hun mainframes uit te rusten met additionele vectorunits, zoals de Vector Facility voor de IBM 3090 en een soortgelijke unit voor de CYBER 18-990 van Control Data. Voor details en snelheidsmetingen voor dit soort opgewerkte mainframes, zie [41].

De laatste jaren is nu ook parallellisme van complete processoren commercieel exploiteerbaar gebleken en meer dan dat: parallellisme en vectorprocessing hebben elkaar duidelijk gevonden. Vectorsupercomputers worden steeds meer uitgebreid met meerder vector CPU's (bijvoorbeeld CRAY X-MP, CRAY-2, ETA 10, IBM 3090/400E-VF) en parallelle machines worden, althans voor wat betreft de topmodellen, gaandeweg uitgerust met vectorunits (bijvoorbeeld INTEL's iPSC). Ook hier is inmiddels de eerste parallelle vectormini verschenen: de Sugarcube van INTEL.

De echte supercomputers van de komende jaren zullen bestaan uit shared memory systemen met een relatief groot aantal krachtige vector CPU's en misschien zal ook een

enkel message passing systeem met een nog groter aantal vector CPU's de status van supercomputer bereiken.

Naar verwachting zullen de shared memory systemen allemaal een vorm van multitasking kennen om parallellisme tot uiting te brengen, van goede automatisch parallelliserende compilers zullen we voorlopig nog wel verstoken blijven. Vermoedelijk zal er wel een vrij uniforme manier voor het tot uitdrukking brengen van parallellisme in FORTRAN komen; we zagen reeds de sterke analogie tussen het EVENTS-mechanisme (paragraaf 14.3) en het send-receive mechanisme voor de message passing systemen (paragraaf 15). Zoiets zal wel moeten ontstaan om codes nog een beetje overdraagbaar te houden en snel te kunnen aanpassen aan een bepaalde machine (belang van zowel fabrikant als gebruiker). De taal FORTRAN is zelf niet zo geschikt om parallel te programmeren en men zou zich kunnen afvragen of het niet een aflopende zaak is met deze taal. Het is misschien aardig op deze plaats Seymour Cray te citeren, die ooit opmerkte: 'Ik weet niet hoe de taal van de toekomst er uit zal zien, maar de mensen zullen het FORTRAN noemen.'

Het laat zich op dit moment al aanzien dat het parallel (her)programmeren van codes niet het grootste probleem gaat worden. We hebben gezien dat vrijwel alle supercomputers gebruik maken van een hiërarchisch geheugen: vectorregisters - cache - centraal geheugen - achtergrondgeheugen. Voor grotere problemen speelt deze hiërarchie een grote rol ten aanzien van de rekensnelheid, zie bijvoorbeeld de voorbeelden voor de IBM 3090 of de Alliant FX/8. In wezen benaderen de vectorregistermachines alleen hun topsnelheid indien men terdege rekening houdt met deze hiërarchie door vectorgedeelten zo lang mogelijk in de vectorregisters te houden en te benutten (via bijvoorbeeld chaining), en ook voor de CYBER 205 hebben we de invloed van het hiërarchisch geheugen gezien in de discussie rond het virtual memory concept in hoofdstuk 10. Naarmate het aantal CPU's toeneemt zal de rol van lokale geheugens en cache geheugens nog beduidend groter worden. Daaruit vloeit voort dat de dataorganisatie en het transport van data tussen geheugens steeds belangrijker worden. Voor de message passing systemen is dat evident (zie ook het voorbeeld voor de NCUBE/4 in paragraaf 15). Het zal blijken dat de efficiëntie van een code bepaald gaat worden door de dataorganisatie (de verdeling van data over de geheugens) en door het vermogen data uit de snelste geheugens zo lang mogelijk te kunnen blijven benutten (om transport te vermijden).

Dit vereist voor vele bestaande codes een ingrijpende herstructurering en zal vele manjaren gaan kosten. Een geluk bij een ongeluk is dat veel van dit soort codes aan een ingrijpende revisie toe zijn en dat deze herstructurering de kans biedt meteen de bezem door vele verouderde algoritmen en technieken te halen. Het is van groot belang dat men er zorg voor draagt de hiervoor benodigde expertise te (helpen) ontwikkelen.

We besluiten dit hoofdstuk met een soort overzicht van de laatst binnengekomen nieuwtjes omtrent recent aangekondigd supercomputers voor de vroege negentiger jaren.

ETA, de supercomputerdochter van CONTROL DATA, heeft inmiddels de eerste exemplaren uit haar ETA10 serie afgeleverd. De ETA10 machines kunnen uit meerdere processoren bestaan, elke processor kan gezien worden als een (vernieuwde en verbeterde) CYBER 205. Het topmodel ETA10-G wordt geleverd met maximaal 8 2-pipe vector

CPU's met een klokcyclus van 7 ns. CONTROL DATA (ETA) claimt een topsnelheid voor deze machine van 10286 Mflops, ofwel ongeveer 10 Gflops. Deze snelheid wordt waarschijnlijk gehaald door in halve precisie paralllel linked triads uit te voeren en gebruik te maken van het feit dat de scalaire en vector functional units binnen elke CPU ook parallel kunnen werken. Op het moment van schrijven is de ETA10-G nog niet voor benchmarking beschikbaar en moeten we het doen met documentatie [52]. Hieruit blijkt dat parallel programmeren van de machine zal gaan volgens Multitasking principes. Ook zal de FORTRAN compiler anticiperen op FORTRAN 8x vector syntax (waarschijnlijk analoog aan de bekende CYBER 205 vector syntax).

Het rekencentrum ENR te Petten beschikt met ingang van september 1988 over een eigen 2-processor ETA 10-P (klokcyclus 24 ns, maximumsnelheid 750 Mflops) [24].

In februari 1988 werd ook de reeds langere tijd verwachte CRAY Y-MP definitief aangekondigd [8]. Het betreft hier een versnelde versie van de CRAY X-MP: maximaal 8 CPU's met een klokcyclus van 6 ns (de maximale snelheid is ongeveer 2.5 Gflops). Voor 1990 staat de CRAY-3 aangekondigd: 16 processoren met elk een klokcyclus van 2 ns (16 Gflops).

Eind 1987 kondigde CRAY de start aan van het ontwerp van de CRAY-4 [6]. Naar verluidt zou deze machine gebaseerd worden op het gebruik van GaAs circuits en de ontwikkeling vindt plaats onder leiding van Seymour Cray zelf. De CRAY-4 gaat bestaan uit 64 CPU's met een klokcyclus van slechts 1 ns (!). Met behulp van chaining zou dat kunnen leiden tot een top van 128 Gflops (als men tenminste weer kans ziet het seriële gedeelte van de code en de synchronisatiekosten tot een uiterst minimum beperkt te houden, maar dat geldt voor alle multiprocessormachines). Vermoedelijk zal men deze machine met Multitasking te lijf kunnen, zodat de gebruiker dan als het ware een gigantisch reservoir van rekencapaciteit ter beschikking staat.

De Volkskrant van 30 januari 1988 meldt dat de supercomputerontwerper Steve Chen de firma CRAY heeft verlaten, na onenigheid over de te volgen lijn, en nu in dienst is getreden van gigant IBM. IBM zou voornemens zijn Chen de gelegenheid te geven een krachtige IBM compatible 64 CPU vectorprocessor te ontwikkelen (als concurrent van de CRAY-4?). Computerdeskundigen speculeren er op dat dit de druk op de andere firma's zal verhogen om spoed te zetten achter parallelle activiteiten. Genoemd worden de namen van Digital Equipment, NEC, Hitachi en Fujitsu. Uit Japan komen geluiden dat de laatstgenoemde drie vectorcomputerfabrikanten inderdaad het parallelle pad zijn opgegaan.

Op het message passing systeem front zien we een steeds massiever parallellisme. NCUBE heeft inmiddels een 1024 processor Hypercube machine, en wat te denken van de FPS-T serie die uit te breiden is tot een 16384 vectorprocessorsysteem. Met een snelheid van 16 Mflops per processor zou dat, theoretisch althans, tot een top van 262 Gflops leiden. Het Duitse Supremum project staat klaar om haar prototype van 128 vectorprocessoren (Cluster systeem) te voltooien. Ook INTEL is doende haar Hypercube machines uit te breiden met meer krachtige (=vector) processoren. Bij deze grootschalige Hypercube systemen is de mogelijkheid aanwezig om de machine desgewenst tijdelijk op te splitsen in Hypercubes van lagere orde, analoog aan het opsplitsen van de ALLIANT FX/8 in kleinere eenheden (paragraaf 14.2). Dat lijkt de aangewezen weg voor

vele toepassingen, want het moet uitgesloten worden geacht dat voor een breed scala van wetenschappelijk rekenwerk 1024 CPU's (bijvoorbeeld) beduidend meer effect kunnen opleveren dan bijvoorbeeld 64 of nog minder (parallelle) CPU's (zie tabel 2 en de discussie in paragraaf 12.2).

Kortom, er staan ons nog boeiende tijden te wachten en het zal nog wel even duren voordat we er achter zijn welk soort machine het meest geëigend is voor een bepaalde klasse problemen, zodat het voor een instituut of onderzoeklaboratorium niet eenvoudig zal zijn een goede keuze voor de (reken)toekomst te maken.

LITERATUUR

[1] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, *Proc. AFIPS Comput. Conf.*, Vol. 30, 1967.

[2] I.Y. Bucher, *The computional speed of supercomputers, The Proc. of ACM Sigmetrics Conf. on Measurements and Modeling of Computer Systems*, 1983.

[3] B. Buzbee, Supercomputers: value and trends, in: A. Lichnewsky & C. Saguez, *Supercomputing*, North-Holland, Amsterdam, 1987.

[4] S.C. Chen, D.J. Kuck & A.H. Sameh, Practical parallel band triangular system solvers, *ACM Trans. on Math. Softw.*, Vol. 4(3), pp.270-277, 1978.

[5] CRAY CAL Assembler Version 1 Reference Manual, Pub. No. SR-0000 *Rev. J.*, pp.3-85, febr. 1983.

[6] *CRAY News*, Minneapolis, 18 nov. 1987.

[7] CRAY Computer Systems Technical Note, *Multitasking user guide*, SN-0222, CRAY Research Inc.

[8] CRAY Product Announcement, *The CRAY Y-MP Computer System*, 9 febr. 1988.

[9] J.J.M. Cuppen, A divide and conquer method for the symmetric tridiagonal eigenproblem, *Numerische Math.*, Vol.36, pp.318-340, 1981.

[10] D. Dent & R. Gibson, Multitasking the ECMWF weather forecasting model, *CRAY Channels*, winter 1988.

[11] J.J. Dongarra, Performance of various computers using standard linear equations software in a FORTRAN environment, Technical Memorandum, 8 dec. 1986, Argonne National Lab., Argonne, USA.

[12] J.J. Dongarra & S.C. Eisenstat, Squeezing the most out of an algorithm in CRAY Fortran, Report ANL/MCS-TM-9, Argonne National Lab., Argonne, 1983.

[13] J.J. Dongarra, J. du Croz, S. Hammarling & R.J. Hanson, A proposal for an extended set of Fortran basic linear algebra subprograms, *SIGNUM 20*, 1985.

[14] J.J. Dongarra & I.S. Duff, Performance of vector computers for direct and indirect addressing in Fortran, Harwell Report, 1986.
[15] J.J. Dongarra, J. du Croz, I.S. Duff & S. Hammarling, A proposal for a set of Level 3 basic linear algebra subprograms, Harwell Report CSS 212, oct. 1987.
[16] J.J. Dongarra & I.S. Duff, Advanced architecture computers, Techn. Mem. No. 57 Revision 1, Argonne National Lab, Argonne, 1987.
[17] J.J. Dongarra, Performance of various computers using standard linear equations software in a FORTRAN environment, Techn. Mem. 23, Argonne National Lab., Argonne, 29 nov. 1987.
[18] J.J. Dongarra & T. Hewitt, Implementing dense linear algebra algorithms using multitasking on the CRAY X-MP-4 (or approaching the Gigaflop), Techn. Mem. No. 55, Argonne National Lab., Argonne, aug. 1985.
[19] J.J. Dongarra, C.B. Moler, J.R. Bunch & G.W. Stewart, *LINPACK User's Guide*, SIAM, Philadelphia, 1979.
[20] I.S. Duff, The use of vector and parallel computers in the solution of large sparse linear equations, Harwell Report, 1986.
[21] I.S. Duff, The influence of vector and parallel processors on numerical analysis, Harwell Report AERE R 12329, 1986.
[22] I.S. Duff, Supercomputing in Europe - 1987, Harwell Report CSS 206, 1987.
[23] I.S. Duff, A.M. Erisman & J.K. Reid, *Direct methods for sparse matrices*, Oxford University Press, London, 1986.
[24] ENR Nieuwsbrief, 1988 no. 1, ENR, Petten.
[25] ETA Systems, ETA10 System Overview, Pub - 1006, Revision A, 28 febr. 1986.
[26] M.J. Flynn, Very High speed computing systems, *Proc. IEEE 14*, 1966, pp.1901-1909.
[27] Fujitsu Facom VP-200, *Vectorization Manual* no. 78SP-5740-1, 1985.
[28] T. Furukatsu, T. Watanabe & R. Kondo, NEC Supercomputer SX System, *Nikkei Electronics*, Vol.11, pp.237-272, 1984.
[29] FX/Series Product Summary, Alliant Computer Systems Corporation, Revised Version, Littleton, Massachusetts, juni 1987.
[30] R. Golliver, B. Hughey & C. Moler, Linpack and Eispack on the Intel iPSC, Application Brief AB-03-687, Intel Scientific Computers, 1987.
[31] D. Heller, Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems, *SIAM J. Numer. Anal.*, Vol.13(4), 1976, pp.484-496.
[32] R.W. Hockney, $(r_\infty, n_{\frac{1}{2}}, s_{\frac{1}{2}})$ measurements on the 2-CPU CRAY X-MP, *Parallel Computing* 2, 1985, pp.1-14.
[33] R.W. Hockney & C.R. Jesshope, *Parallel Computers-architecture, programming and algorithms*, Adam Hilger, Bristol, 1981.
[34] IMSL User's Manual - Math/Library, Version 1.0, IMSL, Houston, april 1987.
[35] W. Jalby, CEDAR Architecture, in: A. Lichnewsky and C. Saguez, *Supercomputing*, North-Holland, Amsterdam, 1987.
[36] T.L. Jordan, A guide to parallel computation and some CRAY-1 experiences, LANL Report LA-UR-81-247, Los Alamos National Lab., Los Alamos, NM, 1981.

[37] A. Karp, A parallel processing challenge, *SIGNUM on Numer. Math.*, Vol. 20[4], 1985, pp.12-13.
[38] A.H. Karp, Programming for parallelism, *IEEE Computer*, mei 1987, pp.43-57.
[39] J.M. van Kats, R. Llurba & A.J. van der Steen, Results of a benchmark test on a Siemens VP-200 vector processor with comparisons to other supercomputers, Report TR-19, Academisch Computer Centrum, Utrecht, 1986.
[40] J.M. van Kats, R. Llurba & A.J. van der Steen, A close look at the first generation of Japanese supercomputers, Report TR-22, Academisch Computer Centrum, Utrecht, 1986.
[41] J.M. van Kats & A.J. van der Steen, Mainframes with supercomputer speed, Techn. Report TR-26, Academisch Computer Centrum, Utrecht, 1987.
[42] J.M. van Kats & A.J. van der Steen, Minisupercomputers - a new perspective?, Report TR-24, Academisch Computer Centrum, Utrecht, 1987.
[43] C.L. Lawson, R.J. Hanson, D.R. Kincaid & F.T. Krogh, Basic linear algebra subprograms for Fortran usage, *ACM Trans. Math. Software*, Vol. 5(3), 1979, pp.308-323.
[44] A.C. McKellar & E.G. Coffman Jr., Organizing matrices and matrix operations for paged memory systems, *Comm. of the ACM*, Vol. 12(3), 1969, pp.153-165.
[45] P.H. Michielse, Domain decomposition in parallel reservoir simulation, Report 87-63, TU Delft, 1987.
[46] P.H. Michielse & H.A. van der Vorst, Data transport in Wang's partition method, *Parallel Computing*, 7, 1988, pp.87-95.
[47] C. Moler, A. closer look at Amdahl's law, Technical note TN-02-687, Intel Scientific Computers, 1987.
[48] NAG Library, Mark 11, NAG, Oxford, dec. 1984.
[49] J.K. Reid, Fortran 8x - the new Fortran standard, Report AERE R 12857, U.K.A.E.A. Harwell, sept. 1987.
[50] J.K. Reid, Sparse matrices, in: D.A.H. Jacobs (ed.), *The state of the art in numerical analysis*, Academic Press, New York, 1977.
[51] J.J.F.M. Schlichting & H.A. van der Vorst, Solving bidiagonal systems of linear equations on the CDC CYBER 205, Report NM-R8725, CWI, Amsterdam, 1987.
[52] R. Schreiber & W.P. Tang, Vectorizing the conjugate gradient method, Report of Dept. of Computer Science, Stanford, CA, 1983.
[53] M. Shimasaki, The FACOM VP100 at the Kyoto University Data Processing Center and vectorization of certain standard constructs for linear algebra computations, *IPSJ Numerical Analysis*, Vol.12(1), 1985, pp.1-10, (in het Japans).
[54] H.S. Stone, Problems of parallel computation, in: J.F. Traub (ed.), *Complexity of sequential and parallel numerical algorithms*, Academic Press, New York, 1973.
[55] P.J. Sydow, Optimization guide SN-0220, CRAY Computer Systems Technical Note, Cray Research Inc., mei 1982.
[56] H. Tamura, Y. Shinkai & F. Isobe, The supercomputer FACOM VP System, *Fujitsu Scientific and Techn. J.*, Vol.21(1), 1985, pp.90-108.
[57] H.A. van der Vorst, (M)ICCG for 2D problems on vectorcomputers, Report A-17, Kyoto University, Kyoto, Japan, 1986.
[58] H.A. van der Vorst & J.M. van Kats, The performance of some linear algebra

algorithms in Fortran on CRAY-1 and CYBER 205 supercomputers, Report TR-17, Academisch Computer Centrum, Utrecht, 1984.

[59] H.A. van der Vorst & K. Dekker, The vectorization of linear recurrence relations, Report 88-21, TU Delft, 1988.

[60] H.H. Wang, A parallel method for tridiagonal equations, *ACM Trans on Math. Softw.*, Vol.7(2), 1981, pp.170-183.

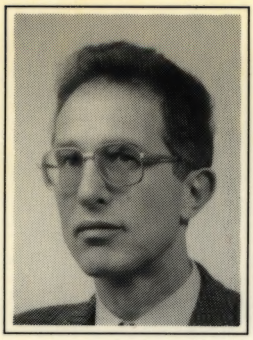
[61] T. Watanabe, Design concept for highspeed vector and scalar processing: architecture of the NEC Supercomputer SX System, Presented at ICCD '86 Conf., 6-9 oct. 1986, Port Chester, N.Y.

[62] J.H. Wilkinson, *The algebraic eigenvalue problem*, Clarendon Press, Oxford, 1965.

[63] D.T. Winter, Efficient use of memory and input/output, in: J.J. te Riele. T.J. Dekker & H.A. van der Vorst (eds.), *Algorithms and applications on vector and parallel computers*, North-Holland, Amsterdam, 1987.

[64] H.A.G. Wijshoff, *Data Organization in parallel computers*, Proefschrift, RU Utrecht, 1987.

INDEX



Over de auteur

De auteur is hoogleraar in de numerieke wiskunde aan de Technische Universiteit Delft. Een belangrijk deel van het onderzoek binnen zijn groep is gericht op de zeer grote stelsels vergelijkingen die ontstaan bij het oplossen van partiële differentiaalvergelijkingen. Sedert 1979 maakt hij daarbij gebruik van supercomputers en dat heeft onder meer geleid tot verschillende publikaties met als thema het realiseren van hoge rekensnelheden bij het oplossen van stelsels vergelijkingen. De auteur heeft verder vele voordrachten gegeven over dit onderwerp en was onder meer, op uitnodiging van de Japanse overheid, enige tijd te gast in Japan voor onderzoek naar de mogelijkheden van Japanse supercomputers. De op uiteenlopende computersystemen opgedane ervaringen hebben de inspiratie geleverd voor dit boek.

Over het boek

De voor grensverleggend onderzoek en adequate modellering vereiste hoge rekensnelheden beginnen binnen bereik te komen door toepassing van verschillende vormen van parallel rekenen. Het boek beoogt een inleiding te zijn tot de mogelijkheden van commercieel beschikbare parallelle en z.g. vector-computers en de lezer een idee te geven van de daarvoor vereiste reken- en programmeertechnieken. De behandeling is in hoofdzaak gericht op problemen in de technisch-wetenschappelijke sfeer. Voor een aantal relatief zeer eenvoudige, doch representatieve problemen wordt getoond hoe deze effectief op de nieuwe geavanceerde computers kunnen worden verwerkt. Het geheel is geïllustreerd met talrijke gegevens over allerlei bestaande supercomputers en de lezer krijgt een indruk van het werkelijke effect van parallel rekenen via praktisch opgedane ervaringen. Mede op basis van deze ervaringen worden ook een paar voorzichtige prognoses gedaan ten aanzien van toekomstig te verwachten ontwikkelingen.

ISBN 90 6233 300 1
NUGI 853